TMIL

Text Manipulation Imaging Language

FINAL REPORT

Eli Hamburger (eh2315@columbia.edu)
Michele Merler (mm3233@columbia.edu)
Jimmy Wei (jw2553@columbia.edu)
Lin Yang (ly2179@columbia.edu)

December 18, 2007

Contents

Chapter 1

White Paper

1.1 Introduction

TMIL (pronounced TEE-mil), short for Text Manipulation Imaging Language, is a revolutionary high level programming language that allows users to manipulate text programatically on an image and even draw on it. Users of the language can generate small programs that can do sophisticated text manipulations on images, without having to resort to complicated graphics libraries or painting programs such as Adobe Photoshop. Text manipulation has a wide range of applications, specifically geared towards web development. Some interesting scenarios include:

- Allowing a user of a Content Management System to display text to website users in fonts that users don't have on their computer
- Allowing a web site template designer to create template that are easily adaptable
- Allowing users to interactively label specific regions of an image.
- Generating CAPTCHAs, an image based challenge-response tests used on many web site registration forms.
- Manipulating text on multiple images to create an animation.

There are image processing libraries available that can manipulate text and draw on images, but they are very difficult and cumbersome to use. TMIL was designed from the ground up to have a clean and simple syntax so that users can do repetitive and complicated imaging tasks quickly and efficiently.

1.2 Motivation and Features

Our primary motivation for creating TMIL is to create a specific purpose language that is easy enough to enable developers of all backgrounds to use, while remaining powerful enough to draw exactly what the programmer wants. The Java Paint2D and the GD2 library for C++ are both powerful, but require a lot of effort for even the simplest projects. Other command based image editing application such as ImageMagick require fine tinkering of command line arguments in ununderstandable order. While the simple MSPaint and expansive Adobe Photoshop offer similar text on image capabilities, they require user interaction. The TMIL Language offers the developer a way to automatic adding text or drawing to images in a construct that feels **natural**.

The specificity of the language also brings with it **security**. Administrators can give free compilation and execution rights for TMIL code and applications knowing the user is limited in his power. The user is constraint to editing images and has no access to other parts of the system through TMIL code.

TMIL uses standard and recognizable constructs such as if statements and while loops. The language also supports native objects such as integers and strings as well as standard functions that are normally available for such objects. The simple tools give a programmer a lot of manipulation ability.

TMIL code flow is **intuitive**, allowing developers to lay down text in the code the same way they would think about doing it interactively. The user sets the properties of the text – such as font, font-size, color etc – he/she wishes to set on the image, and stamps it on. The location of the text and even whether or not to rotate the text are all properties of the text object and can be changed anytime until the text is stamped. This allows the programmer to do what he/she feels most natural. Drawing is also easy yet effective, since the user can specify the extremes of the line, its thickness and color.

TMIL is **flexible**. The user can create one text object and stamp it onto many images, or stamp many text objects sequentially onto a single image. Obviously, he/she can do a combination of both. The user can also easily create any composition of lines he/she wishes.

1.3 Example of Syntax

The simple code example described here loads an image and writes text on it at different positions, with different colors, rotations and fonts. It uses all the built-in types offered by the language, together with some specific functions.

```
int i = 0;
image im;
open("namefileIn.png");   // loads image
text t;
t.name = "dog";              // assigns value "dog" to property string of t
t.font = "Arial.ttf";              // assigns arial font to t
t.size = 60;                   // assigns a size to t
coordinate coor[3];
color c;
c.r = 255;
c.g = c.b = i;              // sets both the g and b values of c to zero

for (i=0; i<=2; i++) {

        c.r = c.r - i*20;    // assigns values to the r, g, and b properties of c
        c.g = c.b + i*60;
        c.b = c.g + i*60;
        t.color = c;                // assigns value c to the color property of t
        coor[i].x = i*150;          // assigns the x and y values of coor
        coor[i].y = 200;
        t.rotation = i*50;      // changes the rotation of t
        t.position = coor[i];   // assigns a value to the position of t
        if(i==1)
            t.font = "Times.ttf";
        im <~ t; // stamps t to the image im
}

// draws a line on im, from position coor[0] to position coor[1],
// with color c and 5 pixels thick

drawline(im, coor[0], coor[1], c, 5);

save(im, "namefileOut.png");          // saves the result
```

Input and output of this sample code are presented in Figure 1.1 (a) and (b).

(a) (b)

Figure 1.1: (a) Input and (b) output of the sample code presented.

Chapter 2

Language Tutorial

2.1 Introduction

TMIL is a programming language intended to write and manipulate text on images, with an additional functionality for drawing on images as well. Syntax and coding style are similar to C/C++ and Java, so that the language is immediate and familiar to the user. TMIL offers some built-in types each of which has properties, just like classes in C++ or Java. Those intuitive properties provide the user a straightforward way to manipulate the features of the text he/she wants to print. Every TMIL program must include a main function (only one function named 'main' is allowed), which must return an int and must either take no parameters or an array of strings.

2.2 "Hello World"

```
/* TMIL allows the same commenting style as C++  */

int main(){                    // TMIL requires a main function

/*************** Regular types *****************/

   int i = 0;                     // We initialize the regular types
   bool b = true;    // These are basically imported from C++
   float f;
```

```
   char char = 'd';
   string name = "namefileIn.png";

/**************** Built-in types ******************/

 color c;              // 1) color: definition
 c.r = 255;            // sets the values of the color properties:
 c.g = c.b = i;        // r , g and b values. Multiple assignments are handled

 coordinate coor[3];         // 2) coordinate: definition (an (x,y) position in an image)

 image im1,im2;                 // 4) image: definition it has 2 properties:
 i = im.h;                      // height(h) and width(w), which are read only

 text t;                        // 5) text: definition;
 t.name = "Hello World";        // assigns value "dog" to property string of t
 t.font = "Arial.ttf";          // assigns arial font to t
 t.size = 60;                   // assigns a size to t

 /**************** Built-in functions 1 ******************/

 open(im1,name);          // loads image from file: .png and .jpg are supported

/************* control flow (for, if, ifelse, while) **********/

for (i=0; i<=2; i++) {

        c.r = c.r - i*20;          // assigns values to the r, g, and b properties of c
        c.g = c.b + i*60;          // each property of the built-in types is accessed
        c.b = c.g + i*60;          // through the '.' operator
        t.color = c;               // assigns value c to the color property of t
        coor[i].x = i*150;         // assigns the x and y values of coor
        coor[j].y = 200;
        t.rotation = i*50;         // changes the rotation of t
```

```
        t.position = coor;         // assigns a value to the position of t

        if(i==1)
            t.font = "Times.ttf";  // sets the font of t: the font path must be specified
        im <~ t; // the stamp operator prints t to the image im
}

/*************** Built-in functions 2 *****************/

// draws a line on im, from position coor[0] to position coor[1],
// with color c and 5 pixels thick

drawline(im, coor[0], coor[1], c, 5);

save(im, "namefileOut.png"); // saves the result into a file: png and jpg are supported

return 0;
}
```

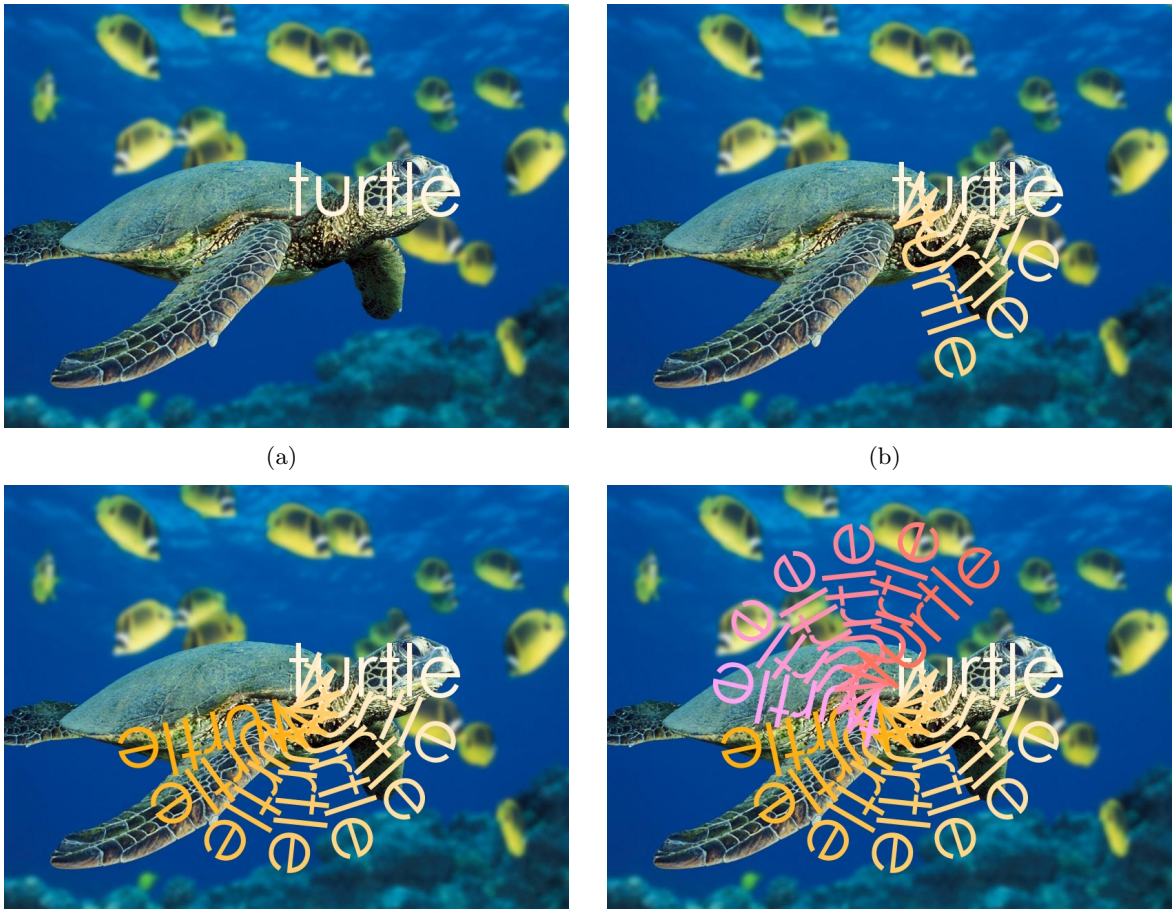
2.3 A more elaborate example

The following code shows that the user can define his own functions and use them, for example, to create a sequence of frames that, when assembled together, can create a text animation.

```
void circular_draw(image im, text w) {             // user defined function
  text w1 = w;
  string name;
  for(int j = 0; j<15; j++) {
     name = "./turtle/turtle" + int2string(j+1) + ".jpg";  // built-in function int2string
     w1.colour.b -= 30;
     w1.colour.g -= 10;
     w1.rotation = - j*(360/15);
      im <~ w1;
```

```
      save(im,name);
    }
}

int main() {

  image im;
  open(im, "turtle.jpg");

  text w1;
  w1.name = "turtle";
  w1.font = "GOTHIC.ttf";
  w1.rotation = 0;
  w1.size = 80;
  w1.position.x = im.w/2;
  w1.position.y = im.h/2;
  w1.colour.r = w1.colour.g = w1.colour.b = 255;

  circular_draw(im,w1);

  return 0;
}
```

Figure 2.1 (a), (b), (c) and (d) shows some of the frames generated by the above program.



(a) (b)

(c) (d)

Figure 2.1: Simple text animation frames generated by TMIL code

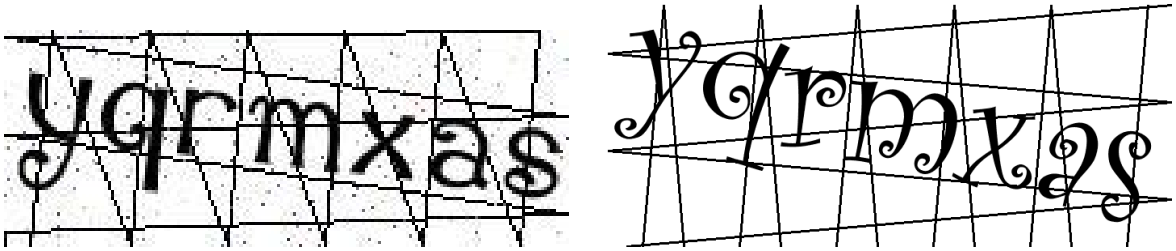
2.4 CATPCHA

CAPTCHA are Remote Turing Tests with which we are challenged every day when we want to register to a website. The following example shows how a user can easily implement a CAPTCHA in TMIL, obtaining for example the same test provided by slashdot.com (Figure 2.2).

```
void zigzag2(image im, color c, int interval){
   coordinate p1,p2;
   int i;
```

```
   p1.x = p2.y = 1;
   p1.y = interval;
   p2.x = im.W -1 ;
   for(i=0;i<im.h/interval;i++){
      if(i%2>0)
        p2.y = p1.y + interval;
      else
       p1.y = p2.y + interval;
   drawline(im,p1,p2,c,3);
   }
   p1.x = 10;
   p1.y = im.w - 1;
   p2.x = p1.x + interval;
   p2.y = 1;
   for(i=0;i<im.W/interval;i++){
      if(i%2>0)
        p1.x = p2.x + interval;
      else
        p2.x = p1.x + interval;
   drawline(im,p1,p2,c,3);
   }
}

int main() {
   color white, black;
   white.r = white.g = white.b = 255;
   black.r = black.r = black.r = 0;
   coordinate p1;
   p1.x = 20;
   p1.y = 130;
   image slashdot;
   create(slashdot,300,700,white);
   text w1;
   w1.font =  "CURLZ___.ttf";
```

```
    w1.colour = black;
    w1.name = "yqrmxas";
    w1.position = p1;
    w1.size = 160;
    w1.rotation = -12;

    slashdot <~ w1;
    zigzag2(slashdot, black, 60);
    save(slashdot, "test3.png");
    return 0;
}
```



(a) (b)

Figure 2.2: (a) slashdot.com original CAPTCHA and (b) TMIL originated CAPTCHA

Chapter 3

Language Reference Manual

3.1 Introduction

TMIL (pronounced TEE-mil), short for Text Manipulation Imaging Language, is a revolutionary high level programming language that allows users to manipulate text programatically on an image. There are image processing libraries available that can manipulate text, but they are very difficult and cumbersome to use. TMIL was designed from the ground up to have a clean and simple syntax so that users can do repetitive and complicated imaging tasks quickly and efficiently.

3.2 Lexical Conventions

3.2.1 Identifiers

An identifier consists of a sequence of upper or lowercase alphabetical characters, numerical digits, and the underscore character. The first character must be an alphabetical character. Identifiers are case sensitive.

3.2.2 Comments

TMIL employs both C and C++ style comments. Multiline comments start with the characters `"/*"` and terminate with the characters `"*/"`. Single line comments can start and end with the above character sequence or start with the characters `"//"` and end at the end of the line.

3.2.3 Reserved Keywords

The following identifiers are reserved keywords in the TMIL language.

```
bool          do        if        true
break         else      image     while
char          elseif    int
color         false     return
continue      float     string
coordinate    for       text
```

3.2.4 Types

The following basic and derived types are supported by the TMIL language.

Type	Description
bool	A basic type that can only have two values, true or false.
char	A basic type of items chosen from the ASCII set
float	A basic type that contains an IEEE 32 bit single precision floating point number, with a range of 1.1 x 10e-38 to 3.4 x 10e38.
int	A basic type that contains a 32 bit signed integer, with a range of -2147483648 to 2147483647.
string	A basic type that contains an arbitrary sequence of characters, surrounded by quotation marks.
color	A built-in type that contains three integer values, each value representing a red, green, and blue value.
coordinate	A built-in type that contains two integer values, to indicate the x and y position of a point in an image
image	A built-in type that allocates enough memory to store an image, with certain properties
text	A built-in type that contains a string of text with certain properties

Built-in Types

Color

Color is a type consisting of three integers, one for each chromatic component of a pixel: red, green and blue. When a color object is created, its properties are all initialized to the value zero.

```
color {
   int r;
   int g;
   int b;
}
```

Coordinate

Coordinate describes the position of a point in the image by its x and y coordinates. When a coordinate object is created, its properties are all initialized to the value zero.

```
coordinate {
   int x;
   int y;
}
```

Image

Image is a type built-in to store an image file. It has two properties: integers h and w representing the height and width of the image itself. When an image object is created, its properties are initialized to the values corresponding to the characteristics of the image loaded or created. Image properties are read only for the user, apart from when he creates an image.

```
image {
   int h;
   int w;
}
```

Text

Text is the most complex built-in type, comprehensive of many properties. It contains a string of characters, and allows the user to set its following properties: size, font, position, color, rotation and name. When a text object is created, its integer properties and subproperties (colour, position, rotation, size) are all initialized to an empty string which yelds the value zero, the font property is initialized to the default font of the system, while name is initialized as an empty string.

```
text {
   string name;
   string font;
   color colour;
   coordinate position;
   int rotation;
   int size;
}
```

3.2.5 Constants

Four types of constants are allowed in the TMIL language. Integer and floating point constants are represented in decimal format (base 10).

Integer Constants

An integer constant consists of a sequence of 1 or more numerical digits.

Floating Point Constants

A floating point constant follows closely with the C convention. It has a fractional or an exponential part and can be expressed in decimal or signed exponent notation. A decimal point without a preceding digit is not allowed.

Char Constants

An char constant consists of a sequence of 1 or more characters.

String Constants

A string constant consists of a sequence of zero or more characters that are surrounded by quotation marks. It cannot include newlines in it. A character escape sequence (\") is required to enclose

quotation marks within strings. Tabs can be inserted using \t, and newlines with \n. "\\" must be used to insert backslash.

3.2.6 Special Characters

Some characters have special significance in the TMIL language:

Special Character	Use	Example
[ ]	Array delimiter	char arr[20];
{}	Function body, or compound statement delimiter	x = 5; x++ ;
( )	Function parameter list delimiter; also used in expression grouping	func();
,	Argument list separator	func2(x, y);
=	Declaration initialize	x = 5;
;	Statement end	x++;
" "	String literal	char str[] = "Hello World";
.	Property access	t.colour.g;

3.3 Conversions

The following conversion are valid between types.

3.3.1 Float to integer

When a floating value is converted to an integral value, the rounded value is preserved as long as it does not overflow.

3.3.2 Integer to float

When an integral value is converted to a floating value, the value is preserved.

3.3.3 Char and integer

A char object may be used anywhere an integral value could be. The char value is converted to an int by propagating its sign through the upper 8 bits of the resulting integer.

3.3.4 Bool to integer or char

A bool value may be converted to an integral of value 0 in the case of false, 1 in the case of true.

3.3.5 Char to string

A char object may be converted to a string of length 1 character, the one expressed by the char value.

3.4 Expressions and Operators

Expressions consist of identifiers and operators. The table below lists the precedence and associativity of all operators in TMIL. Described from highest precedence to lowest.

Token	**Operators**	**Associativity**
Identifiers, constants, string literal, parenthesized expression	Primary expression	
`() [] .`	Function calls, subscripting, property	L/R
`++ -`	Increment, decrement	L/R
`!`	Logical NOT	L/R
`+ -`	Sign operator	L/R
`* / %`	Multiply, divide, modulus after division	L/R
`+ -`	Plus, minus	L/R
`== !=`	Equality comparisons	L/R
`> >= < <=`	Relational comparisons	L/R
`&&`	Logical AND	L/R
`||`	Logical OR	L/R
`=`	Assignment	R
`,`	Comma	L/R
`<~`	stamp	R

3.4.1 Primary expressions

Basic primary expressions

Identifiers :
An identifier is a reference to an object or function. A description of identifiers can be found in Chapter 2.
Constants :
A constant's type is determined by its form and value. Constant expression's type is identical after the operations are performed. A description of constant can be found in Chapter 2.
String literals :
A string literal is a characters array.

```
string literals:          '"' (.)* '"' ;
```

Parenthesized expression:
A parenthesized expression's type is the same as what is parenthesized.

```
parenthesized expression:    '('expression')' ;
```

Subscripts

The element of an array can be accessed by having an index number within square brackets after the array's object identifier. The return value is the same as the type of the array. For example, array[i] returns the ith element of the array array.

```
array element:          ival '['index number']' ;
index number:           digit(digit)* ;
```

Property Operator

An object primary expression followed by a period and the name of a type property can access this property. The return type is the same as the member accessed.

```
type number:            ival'.'(id|ival) ;
```

Function calls

A declared function followed by a pair of parenthesizes with possible variables in between is a function call. The return value is the declared function return type.

```
function call:      id'('parameters')';
parameters:         expression(',' expression)*
                    | E ;
```

3.4.2 Unary operators

Increment and decrement

An object primitive expression followed by double plus signs or double minus signs is increment or decrement. The expression's type can only be int or float. The return type is int.

```
increment:      ival '++' ;
decrement:      ival '-' ;
```

Logical NOT

The logical NOT operator followed by a expression returns the opposite of the expression. The return type is boolean. If the expression's type is int or float, it will return true if the expression's value is zero and return false if the value is not zero. The expression cannot return any type other than int, float or boolean.

```
logical not:        '!' expression ;
```

Sign operator

A plus or minus sign followed by a primitive expression returns 0 plus or minus the expression. The expression's type can only be int or float. The return type is the same as the expression's type.

```
sign operator:      '+ | -' expression ;
```

3.4.3 Arithmetic operators

Multiply and divide

A primitive expression followed by a multiple sign or a divide sign, followed by a primitive expression returns the product or quotient of the two expressions. The expressions' type can only be int or float. The return type is int if both expressions' type are int, otherwise, the return type is float.

`multiply/divide`: *expression* `'*` | `/'` *expression* `;`

Modulus after division

A primitive expression followed by a modulus sign, followed by a primitive expression returns the modulus after the division. The return type is int. If the expression's type is float, it will be truncated before the operation is applied.

`modulus after division`: *expression* `'%'` *expression* `;`

Plus and minus

A primitive expression followed by a plus sign or a minus sign, followed by a primitive expression returns the sum or difference of the two expressions. The expressions' type can be int, float or string. The return type is int if both expressions' type is int. The return type is double if the expressions' type are int and float respectively or both float. The return type is string when both expressions' type are string and the operator is plus. In such situation, the returned value is the concatenated string of the first string and the second string. The expressions' type can only be one of the situations listed above.

`add minus`: *expression* `'+` | `-'` *expression* `;`

3.4.4 Relational operators

Equal

A primitive expression followed by double equal signs, followed by a primitive expression returns true when the two expressions are the same. Otherwise, it returns false. The return type is boolean. If the type of the expressions are int and/or float, the operator will compare the two expressions'

value. If both expressions are string, it will return true when two strings are the same, otherwise it will return false. The expressions' type can also both be boolean. The expressions' type can only be one of the situations listed above.

`equal comparison:` *expression* `'=='` *expression* `;`

Not equal

A primitive expression followed by a not equal sign, followed by a primitive expression returns true when the two expressions are different. Otherwise, it returns false. The return type is boolean. If the type of the expressions are int and/or float, the operator will compare the two expressions' value. If both expressions are string, it will return true when two strings are different, otherwise it will return false. The expressions' type can also both be boolean. The expressions' type can only be one of the situations listed above.

`Not equal comparison:` *expression* `'!='` *expression* `;`

Relational comparisons

Greater than

A primitive expression followed by a greater than sign, followed by a primitive expression returns true when first expression is greater than the second expression. Otherwise, it returns false. The return type is boolean. The expression's type can only be int or float.

`greater than:` *expression* `'>'` *expression* `;`

Not greater than

A primitive expression followed by a not greater than sign, followed by a primitive expression returns true when first expression is not greater than the second expression. Otherwise, it returns false. The return type is boolean. The expression's type can only be int or float.

`not greater than:` *expression* `'<='` *expression* `;`

Less than

A primitive expression followed by a less than sign, followed by a primitive expression returns true

when first expression is less than the second expression. Otherwise, it returns false. The return type is boolean. The expression's type can only be int or float.

`less than`: *expression* `'<'` *expression* `;`

Not less than

A primitive expression followed by a not less than sign, followed by a primitive expression returns true when first expression is not less than the second expression. Otherwise, it returns false. The return type is boolean. The expression's type can only be int or float.

`not less than`: *expression* `'>='` *expression* `;`

3.4.5 Logical operators

Logical AND

A primitive expression followed by a logical AND sign, followed by a primitive expression returns true when both expressions are true. Otherwise, it returns false. The return type is boolean. The expressions' type can only be boolean, int or float. If the expression's type is not boolean, the expression is true when the value is not zero. Otherwise it is false.

`logical AND`: *expression* `'&&'` *expression* `;`

Logical OR

A primitive expression followed by a logical OR sign, followed by a primitive expression returns true when at least one of the expressions is true. Otherwise, it returns false. The return type is boolean. The expressions' type can only be boolean, int or float. If the expression's type is not boolean, the expression is true when the value is not zero. Otherwise it is false.

`logical OR`: *expression* `'||'` *expression* `;`

3.4.6 Assignment Operator

An object primitive expression followed by an assignment sign, followed by a primitive expression will assign the second expression's value to the first expression. If the ival's type is boolean and

the expression's type are int and/or float, ival is false when expression's value is zero, otherwise it's true. If the ival's type is int and expression's type is float, ival will be assigned the value of the expression after truncated. If the ival's type is int or float and the expression's type is boolean, ival will be 1 if the expression is true, otherwise it's false. If the ival is string then the expression must be string, too. The ival and expression's type can only be one of the situations listed above.

`assignment`: *ival* `'='` *expression* `;`

3.4.7 Comma

Comma is used to separate expressions.

`comma`: *expression* `(',' expression )+ ;`

3.4.8 Stamp Operator

An image type expression followed by a stamp symbol, followed by a text type expression will print the string of the text expression on the image file of the image expression, with all the characteristics specified by the text object attributes.

`stamp`: *imageexpression* `<~` *textexpression*

3.5 Declarations

A declaration specifies the interpretation of identifier(s) or function(s). Variables, arrays and functions must be declared before being referred or called.

3.5.1 Variable declaration

One of more variables can be declared in each declaration. Only the same type of variables can be declared in each declaration. The value of the identifier can be assigned to the identifier when being declared. The type of the ival and the expression must be the same.

`variable declaration`:
type ival`('='`*expression*`)?  (',' ival('='`*expression*`)?)*;`

3.5.2 Array declaration

One of more arrays can be declared in each declaration. Only the same type of arrays can be declared in each declaration. A arraylist can be assigned to the identifier when being declared. The type of the arraylist and the ival must be the same. If the size of the arraylist is smaller than the size of the array, the elements with smaller index number will be assigned first. If the arraylist's size is bigger than the size of the array, the exceeded elements in the arraylist will be ignored.

`variable declaration`:
type ival `'['` *size* `']' ( '='` *arraylist*`)?`
`(','` *ival* `'['` *size* `']' ( '='` *arraylist*`)?)*;`

3.5.3 Function declaration

`Function declaration`:	*type id* `'('` *parameters* `')' ;`
`parameters`:	*type* `(','`*type*`)*`
	| *E* `;`

3.6 Statements

In TMIL, statements are usually executed in sequence. There are a few exceptions, specified in the following paragraph.

3.6.1 Expression Statement

Expression statements are the most common statements in the TMIL language. Usually they are assignments or function calls, and take the form of

expression ;

3.6.2 Compound Statements

Compound Statements consist of several statements enclosed in braces, which are considered as a single statement:

{(*statement*;)∗}

3.6.3 Conditional Statement

There are two forms of the conditional statement:

`if` (*expression*) *statement1*
`if` (*expression*) *statement1* `else` *statement2*

In both cases *expression* is evaluated. If it is non-zero, *statement1* is executed; in the second case, if *expression* is zero, then *statement2* is executed. `Else` ambiguity is resolved by connecting the `else` with the nearest elseless `if`.

3.6.4 While Statement

The `while` statement takes the form of

`while` (*expression*) *statement*

Expression is evaluated before the execution of *statement*. If it is non-zero, *statement* is executed. The process is iterated until *expression* evaluates to zero.

3.6.5 For Statement

The for statement takes the form

`for` (*expression1* ; *expression2* ; *expression3*) *statement*

This statement is equivalent to

expression1 ;
`while` (*expression2*) {
    statement
    expression3 ;
}

3.6.6 Return statement

Functions return to their caller via the `return` statement, which takes the form

`return` (*expression*)? ;

Either no value (null) or the value of *expression* is returned to the caller of the function, assuming the function is declared to return a value of matching type. If a function is not declared to return a matching type of *expression*, and *expression* is returned, an error occurs. Similarly, it is an error for a function declared to return null to include *expression* in the return statement.

3.7 Scope Rules

The scope rules in TMIL are very similar to those in C or C++. TMIL adopts static scoping. A variable or function is unavailable (out of scope) until it declared. Functions cannot be nested and therefore cannot be overridden after they are declared. They are declared in the global scope and available until the program completes execution. A variable is available until the end of the block defined by {}), in which it was declared, is reached. In the case of variable declared in the global scope, the variable only goes out of scope on program termination. Nested blocks can access variables defined in parent blocks. If a new variable is declared in a nested block with the same name as a variable from a parent block, said variable was overridden. The nested block will then only have access to the new variable from point of declaration. When the block is closed, the original variable will return to scope.

3.8 Built-in Functions

The TMIL language includes some built-in functions which are available to the user.

3.8.1 Open

This function allows the user to load an image file.

`open(`*image* `Im,` *string* `filename) ;`

3.8.2 Create

Create allows the user to create a new image, without having to rely on preexisting files, by providing the size and the background color. The syntax is the following.

```
create(image Im, int sizex, int sizey, color backgroundcolor) ;
```

3.8.3 Save

This function allows the user to save an image after the processing has been performed.

```
save(image Im, string filename) ;
```

3.8.4 Drawline

Drawline draws a line from point a point p1 to a point p2, with the properties specified by the user.

```
drawline(image Im, coordinate p1,  coordinate p2, color col, int width) ;
```

3.8.5 int2string

Converts an integer to a string.

```
string s = int2string(int x ) ;
```

3.8.6 float2string

Converts a float to a string.

```
string s = float2string(float x ) ;
```

3.8.7 string2int

Converts string to an integer.

```
int x = string2int(string s ) ;
```

3.8.8 string2float

Converts string to an integer, with a minimum and maximum number of digits after the decimal point.

float `x = string2float(`*string* s ,*int* minNum, *int* `maxNum) ;`

3.8.9 char_at

Returns the char located at a particular position in a string.

char `c = char_at(`*string* s ,*int* `position) ;`

3.9 Example

Here is a sample program:

```
void doSomthing(string s1, string s2) {

        image im;

        open(im,s1);

        color col;

        col.r = col.g = col.b = 150;

        text t;

        t.name = s2;

        t.colour = col;

        t.colour.g = 100;
```

```
        t.position.x = 50;
        im <~ t;
}


void main(){

        doSomthing("myImage.jpg","haha!");

}
```

This program would print *haha!* on the image `myImage.jpg` at y=0, x=150 in the color specified in the function.

Chapter 4

Project Plan

4.1 Project Progress Control

4.1.1 Group Discussion and Planning

TMIL group adopted several critical guidelines to ensure that the milestones and deadlines are achieved on time. The most important one is frequent regular meeting. Group members have a group programming meeting takes place every weekend. Everybody reports to the group on what major changes were made to the project regarding his responsible part. And collects feedback from other members. The group programming meeting has been proved very effective since it's the time when group members exchange ideas on things that can not accomplished individually, such as modification of interface and interaction between lexer/parser group and AST group.
The group member communicates via email and phone as well. Team leader sends out "TMIL Progress Record" every week to ensure that everybody is on the same page and doesn't fall behind. Email communication also helps organizing small group meeting. When a team member has free time and want to work in group, he will send an email and other members will respond right away. The smooth communication between group members ensures that we were able to keep to our schedule quite well. Our group was able to figure out which part of the project needed more attention and which part was ahead of schedule. And we can make quick adjustments according to the plan. Project timeline is in sections 4.5.

4.1.2 Development

The development procedure was split into four stages.

Learning Stage: The first stage was from the beginning of the semester till the release of the proposal. Each member learns knowledge of SVN and ANTLR. This stage could be integrated into development stage but it was proved to be a wise decision since everybody became quite familiar with the tools used in the project before even started coding. Thus fewer mistakes were made when we were in the actual development stage. The idea of TMIL was also created during this stage. But it has been modified along the development when we see problems or new features.

Planning Stage: The second stage was from the proposal till the release of the Language Reference Manual. Although Prof. Edwards said that leader should be a dictator, the planning of TMIL was not done by team leader only but by the whole team. Team members give ideas on implementation, details of functionalities and dispatch of the work. Everybody takes one section of the LRM so that he is more familiar with this part. Then we exchanged the writing to other members for error checking. It also helped each member has a clue of the picture of the whole project. After the LRM is finished, everybody is familiar with the whole project and specialized in several aspects of the project.

Development Stage: The third stage is the most important one. It's from LRM to the final exam. The major aspects of the project include lexar, parser, AST, code generation, testing, libraries and documentation. We first divided the team into two groups so that everybody is not working independently. And after the libraries were finished, three members shifted to one group and the other member keeps working on testing and documentation.

Assembling Stage: The last stage is from the final exam till the submission day. All group members meet more frequently to put the code together. Intensive testing was done in this stage. Demo codes and documentations were also finished.

4.1.3 Testing

We utilized an automated tool to perform testing. Test cases for TMIL and CPP were provided by group members to do varies test. The testing focused on for major sections such as syntax, semantics, code generation and compiled code. Chapter 6 is devoted to testing.

4.2 Roles and Responsibilities

Person	Responsibilities
Eli Hamburger	Lexer/Parser, AST, Testing Program, Documentation
Michele Merler	C++ Library design and implementation TMIL Testing, CPP Testing, Demo code, Documentation
Jimmy Wei	Lexer/Parser, AST, TMIL Testing, Documentation
Lin Yang	AST, C++ Library implementation, TMIL Testing, Documentation

4.3 Programming Style

Consistent and standard coding style ensures better understandability amongst the team. It also produces high quality code of the project.

4.3.1 ANTLR Code

When there are "or" lines, they will be listed vertically align one tab to the rule name. When hierarchy exists, it will be tabbed so that it shows obvious distinguish between hierarchies.

```
type: "int"
 | "float"
 | "char"
```

4.3.2 JAVA Code

For this project, we chose to align the starting curly brace at the end of the declaration of the function, and the end brace on the left side of the last line of the block. Each hierarchy is one tab right to the upper level hierarchy.

```
public static void leaveScope() {
currentScope = currentScope.getParent();
}
```

4.4 Project Development Environment

TMIL is a cross platform language. The TMIL project was developed on Linux(ubuntu), Windows (Windows XP, Windows Vista) and Mac OSX. Thanks to the multi-platform attribute of java, TMIL can be used on any platform that can run java files. TMIL requires GD library installed in the system. Which is also a cross platform library. The GD library we used is version 2.0.35. The C++ compiler we used is g++, version 4.0.1. The grammar, lexer, parser and tree walker were implemented via ANTLR, which was recommended by Prof. Edwards. ANTLR greatly helps the team in building compiler related code. The version of ANTLR is ANTLR 3, but we are using ANTLR 2 grammar in TMIL project. The team project system we used is SVN. The version of SVN is Subversion 1.4.5. The other modules in the project for compiler related code is generated via java. The version of java is 1.5.0_13-119. The IDE we used is Eclipse, version 3.3.1.1.

4.5 Project Timeline

As mentioned above, TMIL development is divided into four stages. The whole development starts from the beginning of the semester till the submission of the project.

September 6th: Project team formation
September 14th: Team meet. Agreed on TMIL as the project
September 20th: White paper completion
September 25th: White paper due
September 27th: Discussion and finalizing functionality
October 1st: Discussion and finalizing grammar
October 8th: LRM division
October 14th: SVN set up. LRM completion
October 18th: LRM due. Role/Responsibility division. Start architectural design
November 3rd: Finish lexer, start parser implementation. Finish C++ class library
November 10th: Finish parser. Testing lexer/parser. Finish architectural design
November 17th: Finish lexer/parser testing. Start AST implementation
December 1st: Finish AST. Finish C++ library. Final report division
December 6th: Final exam
December 10th: Group testing parser/lexer, AST. Example and demo code generation
December 16st: Finish final report. Project finished

December 17th: Finish presentation slide. Meet for presentation
December 18th: Final presentation

Chapter 5

Architectural Design

5.1 Architectural Overview

The TMIL compiler consists of three main components: the TMIL lexer, the TMIL parser, and the TMIL tree walker. Valid TMIL code is used as input to the compiler and as output, valid C++ code is generated. This code can then be compiled on any C++ compiler, provided that the FreeType and GD libraries and the TMIL.h source file are provided. An overview of the TMIL compiler architecture is shown in Figure 5.1.

5.2 The TMIL Lexer

The main responsibility of the TMIL lexer is to break the source code file into a series of tokens that can be understood by the TMIL parser. During this stage, we check to see if the source code file can match a series of lexer rules defined by the TMIL grammar. Rules that define identifiers, numbers, comments, and strings are examples of lexer rules.

5.3 The TMIL Parser

Once the tokens have been generated, it is the role of the TMIL parser to ensure that syntax is correct. The TMIL grammar defines language constructs such as rules for expressions, if statements, and for loops. As tokens are read in, the parser tries to match the current token to a grammar rule. It is also at this time that an abstract syntax tree (AST) representation of the source code is created.

The interface between the lexer and the parser is simple. The parser can only understand tokens that are defined by the lexer so grammar rules that are created in the parser must be based off of tokens defined in the lexer.

Our parser creates a very detailed tree with many branches. We found a complex tree easier to deal with when doing semantic analysis and code generation.

5.4 The TMIL Tree Walker

The tree walker traverses the AST created by the parser and checks the semantics of the source code. A tree parsing grammar is specified so that the tree walker knows how to interpret and walk the AST. As the tree walker traverses each node, embedded Java code may be called to check for semantic errors. It is also at this point that the C++ code is generated.
The walker naturally contains many similar rules to the parser. However, since the tree is already constructed with node labeled, the possible routes that we need to code are straightforward. It is safe to say that at this stage, the only tasks remaining is the static semantic analysis and the code generation.
To keep the interface between the parser and the walker simple, the rules for the walker are similar to the rules in the parser. This is because we want the tree walker to walk the AST precisely as to how the AST was generated.
We chose this particular route because it makes reading the code easier. Also, if we had to make a fix to the grammar for the parser, then the corresponding change in the tree walker is trivial.
During each stage of the compiling process, if an error is detected, the compiler immediately terminates with an appropriate error message

5.5 Symbol Table

The walker uses a static class named Env - short for environment - to handle the static semantic analysis. It encapsulates all symbol table related functionality, such as enter/leave scope and get/put variable/function. It also contains functions to check if type can be coerced/promoted into other types and when operations are allowed between types. The walker calls Env's functions as needed throughout the walker grammar file. Built in functions are defined in a Env as well.
The symbol table is a simply linked list of hashmaps. Each node in the table contains a link to the parent node (null if topmost node) and a map that maps identifier and function names to their cor-

responding types. Since a function and variable cannot share the same name, we use the same table to store information about both. He handle types in a very object oriented manner. Below is a brief description of some of the types and some of the more interesting details about each. Minor note: the map actually maps name -> list of Type objects. For a variables, the list may only contain one element. However, a function can be overloaded and therefor the list may contain multiple elements.

Type: This is the root in the class tree, it contains no information but allows both functions and variables to be stored in the same map.
Children: TypeFunction, TypeVariable

TypeVariable: Parent class for all variables including arrays and class like variables. Contains purely virtual function 'matches' which checks a given TypeVariable instance matches the calling instance's class type. Additionally, each TypeVariable child contains a getCppCode() function that returns the name of the type used in c++. This allows for easy conversions. For example, in TMIL, we have 'text' objects. In c++, 'text' objects and really instances of the 'word' class.
Children: TypeInt, TypeBool, TypeChar, TypeString, TypeArray, TypeClass

TypeInt/Bool/String/Char: There is one class per native type.
Children: None

TypeClass: Abstract type for other type that contain properties. Properties are represented as TypeProperties.
Children: TypeArray (for length property), TypeImage, TypeCoordinate, TypeColor, TypeText

TypeProperty: This does not extend TypeVariable as it fails the 'isa' test. A type property contains three pieces of information: a TypeVariable for the parameters type, a boolean representing whether a the parameter is optional and a cppName for c++ code generation. The last is needed is things like 'array.length' which is translated into 'array.size()'.
Children: None

TypeArray: This represents a single level in an array. The TypeArray instance will contain an internal TypeVariable represent ion that type of array it is. For example, the an int[] would be a TypeArray with an internal type on TypeInt. Since the internal type is a TypeVariable, TypeArray's can be nested allowing for multi-level arrays (I.e. int[][]). The array's matches() function is

overloaded to ensure that two type arrays only match if their internal TypeVariables match. This class also contains special functions to aid in the conversion between [] definition and initialization for arrays in tmil and the templated stl vector notations in c++.
Children: None

TypeImage/Text/Coordinate/Color: Represents the types for images, texts, coordinate and colors (respectively).
Children: None

TypeFunction: Functions contain a TypeVariable return type and an ordered list of TypeParameters. TypeParameters may be optional and coded logic ensures that optional parameters are not followed by required parameters. The 'addParameter' function is used to add a parameter to a type function when declaring the function. Since multiple functions can share the same name, this class contains functionality to ensure a possible new version of a function isn't too similar such that it would cause ambiguity when it come time to choose a version of said function when a function call is made. It also contain functionality to check if the given TypeFunction is compatible with a list of given parameter TypeVariable's from a function call.
Children: None

Final Notes: The structure were chosen to make the symbol related code in the walker as simple and understandable as possible. The structure also naturally allow for expansion in the future, whether a future implementation contains more built in classes/functions or if full functioned user created classes are allowed.

5.6 Code generation

Code is generated within the walker. A StringBuilder Java object, serving as a buffer containing the output code, is available to all functions in the class. Most of tree walker functions therefore write code directly to the buffer as they are called. Since we are parsing the tree depth first and since the tree retains a lot of the structure of the original c++ like TMIL code, writing directly in a single pass is very natural. However, certain functions/branches, such as all expressions, return their c++ code to the function caller (parent node) as to allow the caller to put the code in the perfect place without constant and confusing jumping between ANTLR and Java code. Most of the code generated is very similar to the c++ like TMIL code passed into the compiler. There are a

few differences. The code generator, aided by the Type objects, will convert properties and types from their TMIL version to their c++ version as described above. Also, we have decided represent all arrays in TMIL as stl vectors in c++. Using vectors offers several advantages over arrays. The biggest advantage is that it allows programmer to create arrays whose size is unknown at compile time (i.e. 'int a[n+3];'). They allows us to compute the length of an array easily, allows for arrays to be passed by value, and overloads the '=' operator to copy the array instead of just copying the pointer to the array. The tricky part for the conversion was generating initialization code for the vectors to size each dimension appropriately. The TypeArray class reduced the work significantly and it ends up being a few lines in Java resulting in a single lines initialization in c++. The final code is then placed in between header and footer code generated by the Env class. The header code simply includes the TMIL c++ library. The footer code contains the c++ 'main' function. As mentioned in the LRM, the function named 'main' declared in a TMIL program can have a String[] parameter defined, which is a vector<String> in the c++ code. Therefore, the code generator inserts the code for the c++ main function that converts the (int argc, char* argv[]) parameters into a (vector<string>) and then called the main function declared by the TMIL code. To avoid naming clashes between the c++ code for the 'main' function created by the user and the 'main' appended in the footer, the user's 'main' is written as '_main'. Since IDENTIFIER's cannot start with a '_' there is no chance of a conflict with another function.

5.7 TMIL.h

TMIL.h is C++ file which constitutes the library used to interface the output TMIL code with GD and FreeType. It contains 4 classes: color, coordinate, word (the equivalent of the TMIL text) and image. It also provides methods to access every variable member of each class and modify them. Furthermore, it provides code for the built-in functions: drawline, save, create, open, int2string, float2string, string2int, string2float and char_at.

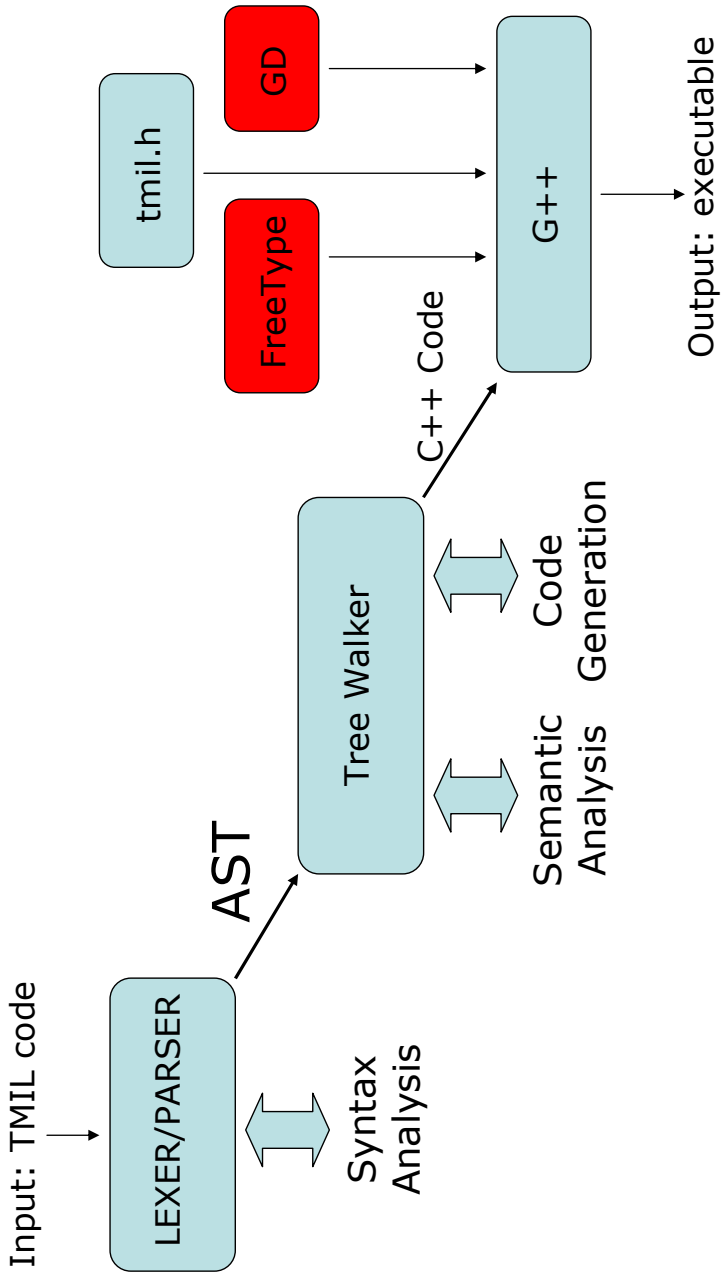


Figure 5.1: Architecture of the TMIL compiler

Chapter 6

Test Plan

6.1 Overview

During the implementation of TMIL, the designers of the language wanted to make sure that programming bugs created during the software development process were caught as early as possible. Thus after each major milestone, we came up with various test cases to ensure that our code was robust and bug free.
To facilitate this process, we focused our efforts on partitioning the test cases into four main categories:

- Lexer Tests
- Parser Tests
- Walker/Semantic Analysis Tests
- Final Programs (more complex and actually do something)

We placed a particular emphasis on testing all aspects of the TMIL Language Reference Manual. For each category, we came up with test cases that were valid and invalid. Tests that were valid ended with a ".good" extension and test cases that were invalid ended with a ".bad" extension.

Each test was placed in their own individual folder named after their test name. The golden output had a ".correct" extension. A bad test would not generate any output.
Three representative test cases are attached at the end of this section. For a specific list of cases that we tested, see the "Coverage" section.

6.1.1 Test example 1

```
file array_acces2.bad

// array access problem : non integer index

int main()
{
float e[10];
coordinate p1;
p1.x = p1.y = 2;
e[p1] = 2.5;
return 0;
}
```

6.1.2 Test example 2

```
file fun_error1.bad

// function error 1 : missing return type

foo(int x){return x};

int main() {
  int a, b, c;
  a = foo(3);
  return 0;
}
```

6.1.3 Test example 3

```
file control_flow.good

// test control flow
```

```
int foo(int x) {
  return x - 1;
}

int main(){

  bool aBool ;
  char aChar;
  color aColor ;
  float aFloat, bFloat ;
  coordinate aCoor, bCoor;
  string aString, bString;
  text aText;
  image aImage;
  int aInt, bInt, cInt;
  text arr[3];
  aInt = 2;

  // nested if
  if(aInt) {
    aBool = 0;
  }
  else {
    if(aInt>3){
       continue;
    }
  }

  // while and call to function
  while(aInt>0){
    aInt = foo(aint);
    if(aInt==1){
       break;
    }
```

```
  }

  // for
  for(int x=3;x<5;x++){
    aInt++;
  }
  for(int y=3;y!=5;y=y+2) {
    aInt++;
  }
  return 0;
}
```

6.2 Automation

To ensure that the designers of the language would test their code changes before making a check in, we placed a particular emphasis on automation and ease of use. Eli created a Java program that would automatically launch a regression when running his program.
Four applications were created for automated testing (listed below). They each take a directory as the first parameter. See the list below for information about other parameters The programs will run the test on every file in the directory. Since some of the file are supposed to ensure the failure (ie. make sure the Lexer does not allow a poorly formatted string) we have used the following convention for denoting when a failed or pass is expected: Files that should pass contain ".good" in their filenames, which files that should fail contain ".bad" in the file name. Files is a directory without a ".good" or ".bad" are skipped. The programs print reports to std out that can easily be parsed by a script should that be needed.

Programs:

- LexerTester - for testing only the lext
- ParserTester - for testing the parser
- TMILTester - for static sematic analysis and code generation testing. Takes a boolean as second parameter to denote verbose mode, which displays detailed errors messages

- GccTester - for testing generated c++ against g++ to ensure they build. Takes a string as second parameter which is the suffix the g++ command to allow users to set library paths.

An example output of the program would look like this:

27 entries found. testing...
1: Testing 'array_access1.bad.txt'. Expected: fail Actual: fail Test: OK
2: Testing 'control_flow3.bad.txt'. Expected: fail Actual: PASS Test: BAD
3: Testing 'control_flow.good.txt'. Expected: pass Actual: pass Test: OK
4: Testing 'fun_error2.bad.txt'. Expected: fail Actual: fail Test: OK
5: Testing 'nested_expressions.good.txt'. Expected: pass Actual: pass Test: OK
6: Testing 'declarations1.bad.txt'. Expected: fail Actual: fail Test: OK
7: Testing 'array_access2.bad.txt'. Expected: fail Actual: PASS Test: BAD
8: Testing 'control_flow1.bad.txt'. Expected: fail Actual: PASS Test: BAD
9: Testing 'fun_error1.bad.txt'. Expected: fail Actual: fail Test: OK
10: Testing 'mismatch_parenthesis_3.bad.txt'. Expected: fail Actual: fail Test: OK
11: Testing 'mismatch_parenthesis_2.bad.txt'. Expected: fail Actual: fail Test: OK
12: Testing 'declarations.good.txt'. Expected: pass Actual: pass Test: OK
13: Testing 'mismatch_parenthesis_1.bad.txt'. Expected: fail Actual: fail Test: OK
14: Testing 'global_variables.good.txt'. Expected: pass Actual: pass Test: OK
15: Testing 'declarations_plus_assignments.good.txt'. Expected: pass Actual: pass Test: OK
16: Testing 'fun_declaration_plus_overloading.good.txt'. Expected: pass Actual: pass Test: OK
17: Testing 'missing_element_2.bad.txt'. Expected: fail Actual: PASS Test: BAD
18: Testing 'declarations3.bad.txt'. Expected: fail Actual: fail Test: OK
19: Testing 'array_access3.bad.txt'. Expected: fail Actual: fail Test: OK
20: Testing 'declarations2.bad.txt'. Expected: fail Actual: PASS Test: BAD
21: Testing 'array_access.good.txt'. Expected: pass Actual: pass Test: OK
22: Testing 'fun_error3.bad.txt'. Expected: fail Actual: fail Test: OK
23: Testing 'operators_2.good.txt'. Expected: pass Actual: pass Test: OK
24: Testing 'control_flow2.bad.txt'. Expected: fail Actual: fail Test: OK
25: Testing 'missing_element_1.bad.txt'. Expected: fail Actual: fail Test: OK
26: Testing 'operators_1.good.txt'. Expected: pass Actual: pass Test: OK
Summary: 26 tests total 21 passed 5 failed
Done testing.

As you can see above, two tests failed in this regression. The designer can then be able to look at the parser1.bad and parser2.bad source code and see a description of the test to get an understanding of why the test failed.
Although this does not prevent a negligent designer from checking in bad code (obviously he can bypass this mechanism without running the regression program), this process did not cause any major problems with the team and when designers did check in bad code, only a handful of tests failed and the code was quickly fixed or the tests updated.

6.3 Coverage

We placed an emphasis on the following scenarios:
Lexer:

- Identifiers
  - Valid and invalid identifier names
- Strings
  - Valid and invalid strings, escaped strings
- Floating point numbers
  - Valid and invalid floating point numbers
- Comments
  - C/C++ style and invalid comments

Parser:

- Declarations
  - Variables
    - With and without arrays
    - With and without assignment
  - Functions

        * With and without parameters
        * With and without return types
    - Global

- Statements
    - Conditional
        * Syntax of if and else, with optional else
    - Control
        * Syntax of for and while loops
    - Expressions
        * Arithmetic, unary, properties, nested, etc.
    - Function Calls
        * Appropriate syntax of function calls

Walker/Semantic Analysis:

- Type Checking
    - Assignments
    - Return types
    - Type coercion
    - Operator type mismatches
- Functions
    - Arguments
    - Return type
    - Default parameters
    - Built in functions
- Precedence
    - Arithmetic precedence

- Scoping
  - Declaring variables inside of a scope
  - Accessing variables inside or outside of a scope
  - Scope for conditionals and loops
  - Nested scopes
- Special Cases
  - Break/continue statement must be within a loop construct
  - Properties and types
  - Assigning values to read only properties (e.g. image height, width)
  - Unrecognized identifiers

6.4 Responsibilities

Eli Hamburger - Wrote a Java program to automate and facilitate the testing process.
Eli Hamburger, Michele Merler, Jimmy Wei, Lin Yang - Brainstormed and came up with individual test cases for each of the categories mentioned above.

Chapter 7

Lessons Learned

7.1 Eli

Before working on this project I never understood how much of a decision making process writing a programing language was. Nothing can be left in the air. Everything, except for a few implementation details, must be decided before completing the compiler. I learned that tools like ANTLR aid in this process significantly as a deviation from certain rules will yield an immediate error. While they may have been annoying while trying to get our Parser to work, they saved us a lot of trouble in the end by ensure an accurate and distinct tree for our treewalker.

As mentioned in our presentation, we accomplished a lot more on this project when working together as a group, as in working at the same place at the same time. We were able to test as we programmed and each person was able to fix bugs very quickly in the areas they knew best. For a complex project such as this, team programming is the best path.

7.2 Michele

Working on this project has thought me what team work means. Using the divide add conquer approach of splitting the work has proved useful, but sometimes it was also important to take a look at each other's work (many silly bugs cannot be seen by the person writing the code, but are easily recognizable for another one). I found the e-mail exchanges very useful, so that anyone making any progress on a task or coming up with a new issue could lead the others to think about it during the week before meeting. Good communication was fundamental.

Working with others also pushes you to write universally understandable code, and not just something that works only for yourself. Furthermore, many times during this semester I found myself learning from people as well as, if not even more than learning from books. I found this kind of practical learning definitely funnier and not less effective!
I particularly appreciated the liberty we had to choose and design our own language. Even though problems, bugs and limited time forced us to reduce the initially intended features of our language, it was particularly rewarding to get it to work and do something original and personal.

One final comment on the timing: the time spent working on a project such as this one is directly proportional to the ambitions of your language, so start as early as possible!

7.3 Jimmy

During the development of the TMIL language and compiler, I learned that it is important to work together as a team. Writing a compiler is a huge project and one person cannot be able to do all of it. Having weekly meetings and working together allowed us to hit our objectives, work efficiently, and deliver a quality product.

One of the biggest difficulties that I personally had was understanding the ANTLR syntax. Although the PLT slides that Professor Edwards presented in class were very useful, the ins and outs of ANTLR were still confusing to me. Ultimately, I had to do additional research online and I found some useful websites such as:

http://tech.puredanger.com/2007/01/13/implementing-a-scripting-language-with-antlr-part-1-lexer/
http://tech.puredanger.com/2007/01/15/antlr-2/
http://tech.puredanger.com/2007/01/17/antlr-3/

However, to get a better understanding of what ANTLR allows and does not allow, one really has to play around with it, especially with the nuances of the tree parser. Thus, future teams should spent ample time understanding ANTLR.

Another lesson that I learned was to keep the grammar as simple as possible. Our team had to modify some grammar rules because of how complex they were. I would remind other teams that

this is a semester long project and prior to developing a language, keep deadlines in check. I cannot stress this enough: Remember to start early!

With careful planning and hard work, I believe future teams will find compiler development interesting and rewarding.

7.4 Lin

I think other members must have mentioned "start early". It's always a good idea to start early on a project. Professor Edwards mentioned this since the first class, which I considered as the most valuable advice. Effective teamwork is also very important. We communicate smoothly through email and phone so that everybody knows exactly what the whole team is up to. SVN is very helpful when working on project in group. Last but not least, it's great to have great teammates.

Chapter 8

Appendix

```
1  /*****************************
2  * Directory: src/tmil
3  *****************************/
4
5
6  /********
7  * File: GTMIL.java
8  ***********/
9  package tmil;
10
11 import java.io.BufferedReader;
12 import java.io.BufferedWriter;
13 import java.io.File;
14 import java.io.FileNotFoundException;
15 import java.io.FileReader;
16 import java.io.FileWriter;
17 import java.io.IOException;
18 import java.io.InputStreamReader;
19 import java.io.Reader;
20
21 import symbolTable.SymbolException;
22 import antlr.CommonAST;
23 import antlr.RecognitionException;
24 import antlr.TokenStreamException;
25
26 public class GTMIL {
27
28         /**
29          * @param args
30          */
31         public static void main(String[] args) {
32                 System.out.println("TMIL Compiler - 2007\n");
33
34                 if (args.length != 1) {
35                         System.err.println("source file must be pasted as a
   parameter");
36                         System.exit(1);
37                 }
38                 try {
39                         String filename = args[0];
40                         String output_name;
41                         int period_index = filename.lastIndexOf('.');
42                         if (period_index > 1)
43                                 output_name = filename.substring(0, filename.length()
44                                                 - period_index + 1);
45                         else
46                                 output_name = filename;
47
48                         System.out.println("Parsing TMIL code...");
49
50                         // Lex, Parse, Walk the code
51                         Reader reader = new BufferedReader(new FileReader(filename));
52                         TMILLexer lexer = new TMILLexer(reader);
53                         TMILParser parser = new TMILParser(lexer);
54                         parser.program();
55                         // Parse the input expression
```

```
56                          CommonAST t = (CommonAST) parser.getAST();
57                          TMILWalker walker = new TMILWalker();
58                          // Traverse the tree created by the parser
59                          String cpp_program = walker.program(t);
60
61                          File temp_file = new File(output_name + ".cpp");
62                          if (temp_file.exists()) {
63                                  temp_file.delete();
64                          }
65                          temp_file.createNewFile();
66                          BufferedWriter temp_file_writer = new BufferedWriter(
67                                          new FileWriter(temp_file));
68                          temp_file_writer.write(Env.generateHeader());
69                          temp_file_writer.write(cpp_program);
70                          temp_file_writer.write(Env.generateFooter());
71                          temp_file_writer.flush();
72                          temp_file_writer.close();
73
74                          String gcc_suffix = " -L\"/u/student/eh2315/lib_dir/libft/lib
    \" -L\"~eh2315/lib_dir/libgd\" -lgd";
75                          String gcc_command = "g++ "+output_name + ".cpp -o "+
    output_name + gcc_suffix;
76                          System.out.println("Invoking g++...");
77                          System.out.println(gcc_command);
78                          /*
79                           * Code to invoke external process adapted from:
80                           * http://www.rgagnon.com/javadetails/java-0014.html
81                           */
82
83                          Process p = Runtime.getRuntime().exec(gcc_command);
84                          String line;
85                          BufferedReader gcc_output = new BufferedReader(
86                                          new InputStreamReader(p.getErrorStream()));
87                          while ((line = gcc_output.readLine()) != null) {
88                                  System.out.println(line);
89                          }
90                          gcc_output.close();
91
92                          if(p.waitFor() == 0)
93                                  System.out.println("Done. " + output_name +
    "created");
94                          else
95                                  System.out.println("G++ Errors.");
96
97                  } catch (FileNotFoundException e) {
98                          System.err.println("File not found.");
99                  } catch (IOException e) {
100                         System.err.println("File io error: " + e.getLocalizedMessage
    ());
101                 } catch (SymbolException e) {
102                         System.err.println("TMIL error: " + e.getLocalizedMessage());
103                 } catch (InterruptedException e) {
104                         System.err.println("G+ runtime error: " +
    e.getLocalizedMessage());
105                 } catch (RecognitionException e) {
106                         System.err.println("TMIL error: " + e.fileName+':'+e.line
```

```
      +'.'+e.column+"> "+ e.getLocalizedMessage());
107                   } catch (TokenStreamException e) {
108                           System.err.println("TMIL error: " + e.getLocalizedMessage());
109                   }
110           }
111   }
112
113   /********
114   * File: Env.java
115   ***********/
116   package tmil;
117
118   import java.util.HashMap;
119   import java.util.List;
120   import java.util.Vector;
121
122   import symbolTable.*;
123   import symbolTable.varTypes.*;
124
125   /**
126    * Static class for storing info
127    *
128    */
129   public class Env {
130
131           public static final TypeVariable TypeInt = new TypeInt(),
132                           TypeFloat = new TypeFloat(), TypeVoid = new TypeVoid(),
133                           TypeChar = new TypeChar(), TypeBool = new TypeBool(),
134                           TypeString = new TypeString(), TypeColor = new TypeColor(),
135                           TypeCoordinate = new TypeCoordinate(), TypeImage = new
      TypeImage(),
136                           TypeText = new TypeText();
137
138           private static final int ARITH = 512; // -,*,/,%
139           private static final int PLUS = 1024; // +
140           private static final int EQUAL = 1536; // ==
141           private static final int COMPR = 2048; // <,<=,>,>=
142
143           private static SymbolTable currentScope, globalScope;
144           private static StringBuilder cppCode;
145           private static HashMap<String, TypeVariable> variableTypes;
146           private static HashMap<String, Integer> variableIndexTable;
147           private static HashMap<Integer, TypeVariable> expressionTable;
148
149           // initialization of types
150           static {
151                   globalScope = currentScope = new SymbolTable(null);
152                   cppCode = new StringBuilder();
153
154                   // ADD VARIABLE TYPES
155                   variableTypes = new HashMap<String, TypeVariable>();
156                   variableTypes.put("void", TypeVoid);
157                   variableTypes.put("int", TypeInt);
158                   variableTypes.put("float", TypeFloat);
159                   variableTypes.put("char", TypeChar);
160                   variableTypes.put("bool", TypeBool);
```

```
161                 variableTypes.put("string", TypeString);
162                 variableTypes.put("color", TypeColor);
163                 variableTypes.put("coordinate", TypeCoordinate);
164                 variableTypes.put("image", TypeImage);
165                 variableTypes.put("text", TypeText);
166
167                 // ADD EXPRESSION TABLE
168                 variableIndexTable = new HashMap<String, Integer>();
169                 variableIndexTable.put("void", 1);
170                 variableIndexTable.put("int", 2);
171                 variableIndexTable.put("float", 4);
172                 variableIndexTable.put("char", 8);
173                 variableIndexTable.put("bool", 128);
174                 variableIndexTable.put("string", 64);
175                 variableIndexTable.put("color", 32);
176                 variableIndexTable.put("coordinate", 16);
177                 variableIndexTable.put("image", 1);
178                 variableIndexTable.put("text", 1);
179
180                 expressionTable = new HashMap<Integer, TypeVariable>();
181                 expressionTable.put(ARITH + 4, Env.TypeInt); // int int
182                 expressionTable.put(ARITH + 6, Env.TypeFloat); // int float
183                 expressionTable.put(ARITH + 8, Env.TypeFloat); // float float
184                 expressionTable.put(ARITH + 10, Env.TypeInt); // int char
185                 expressionTable.put(ARITH + 16, Env.TypeInt); // char char
186                 expressionTable.put(ARITH + 32, Env.TypeCoordinate); // coord coord
187                 expressionTable.put(PLUS + 4, Env.TypeInt); // int int
188                 expressionTable.put(PLUS + 6, Env.TypeFloat); // int float
189                 expressionTable.put(PLUS + 8, Env.TypeFloat); // float float
190                 expressionTable.put(PLUS + 10, Env.TypeInt); // int char
191                 expressionTable.put(PLUS + 16, Env.TypeInt); // char char
192                 expressionTable.put(PLUS + 32, Env.TypeCoordinate); // coord coord
193                 expressionTable.put(PLUS + 64, Env.TypeColor); // color color
194                 expressionTable.put(PLUS + 72, Env.TypeString); // string char
195                 expressionTable.put(PLUS + 128, Env.TypeString); // string string
196                 expressionTable.put(COMPR + 4, Env.TypeBool); // int int
197                 expressionTable.put(COMPR + 6, Env.TypeBool); // int float
198                 expressionTable.put(COMPR + 8, Env.TypeBool); // float float
199                 expressionTable.put(COMPR + 10, Env.TypeBool); // int char
200                 expressionTable.put(COMPR + 16, Env.TypeBool); // char char
201                 expressionTable.put(EQUAL + 4, Env.TypeBool); // int int
202                 expressionTable.put(EQUAL + 6, Env.TypeBool); // int float
203                 expressionTable.put(EQUAL + 8, Env.TypeBool); // float float
204                 expressionTable.put(EQUAL + 10, Env.TypeBool); // int char
205                 expressionTable.put(EQUAL + 16, Env.TypeBool); // char char
206                 expressionTable.put(EQUAL + 128, Env.TypeBool); // string string
207
208                 // ADD BUILT IN FUNCTIONS to function scope
209                 try {
210                         TypeFunction open = new TypeFunction(Env.TypeVoid);
211                         open.addParameter("image", Env.TypeImage, false);
212                         open.addParameter("filename", Env.TypeString, false);
213                         putFunction("open", open);
214
215                         TypeFunction create = new TypeFunction(Env.TypeVoid);
216                         create.addParameter("image", Env.TypeImage, false);
```

```
217                         create.addParameter("width", Env.TypeInt, false);
218                         create.addParameter("height", Env.TypeInt, false);
219                         create.addParameter("color", Env.TypeColor, false);
220                         putFunction("create", create);
221
222                         TypeFunction save = new TypeFunction(Env.TypeVoid);
223                         save.addParameter("image", Env.TypeImage, false);
224                         save.addParameter("string", Env.TypeString, false);
225                         putFunction("save", save);
226
227                         TypeFunction drawline = new TypeFunction(Env.TypeVoid);
228                         drawline.addParameter("image", Env.TypeImage, false);
229                         drawline.addParameter("coordinate1", Env.TypeCoordinate,
    false);
230                         drawline.addParameter("coordinate2", Env.TypeCoordinate,
    false);
231                         drawline.addParameter("color", Env.TypeColor, false);
232                         drawline.addParameter("int", Env.TypeInt, false);
233                         putFunction("drawline", drawline);
234
235                         TypeFunction char_at = new TypeFunction(Env.TypeChar);
236                         char_at.addParameter("string", Env.TypeString, false);
237                         char_at.addParameter("int", Env.TypeInt, false);
238                         putFunction("char_at", char_at);
239
240                         TypeFunction int2string = new TypeFunction(Env.TypeString);
241                         int2string.addParameter("int", Env.TypeInt, false);
242                         putFunction("int2string", int2string);
243
244                         TypeFunction float2string = new TypeFunction(Env.TypeString);
245                         float2string.addParameter("float", Env.TypeFloat, false);
246                         putFunction("float2string", float2string);
247
248                         TypeFunction string2float = new TypeFunction(Env.TypeFloat);
249                         string2float.addParameter("string", Env.TypeString, false);
250                         putFunction("string2float", string2float);
251
252                         TypeFunction string2int = new TypeFunction(Env.TypeInt);
253                         string2int.addParameter("string", Env.TypeString, false);
254                         putFunction("string2int", string2int);
255
256                         // ... this needs to be finished for drawline and the
    int2string...
257                         // functions
258
259                 } catch (SymbolException e) {
260                         e.printStackTrace();
261                 }
262
263         }
264
265         public static void enterScope() {
266                 currentScope = new SymbolTable(currentScope);
267         }
268
269         public static void leaveScope() {
```

```
270                     currentScope = currentScope.getParent();
271             }
272
273             public static void putVariable(String identifier, String type_name)
274                             throws SymbolException {
275                     putVariable(identifier, getType(type_name));
276             }
277
278             public static void putVariable(String identifier, TypeVariable t)
279                             throws SymbolException {
280                     currentScope.putVariable(identifier, t);
281             }
282
283             public static TypeVariable getVariable(String identifier)
284                             throws SymbolException {
285                     return currentScope.getVariable(identifier);
286             }
287
288             public static void putFunction(String identifier, TypeFunction function)
289                             throws SymbolException {
290                     globalScope.putFunction(identifier, function);
291             }
292
293             public static TypeFunction getFunction(String identifier,
294                             Vector<TypeVariable> passed_parameters) throws
    SymbolException {
295                     return globalScope.getFunction(identifier, passed_parameters);
296             }
297
298             public static TypeVariable getType(String type_name) throws SymbolException {
299                     TypeVariable t = variableTypes.get(type_name);
300                     if (t == null) {
301                             throw new SymbolException("Unknown Type: " + type_name);
302                     }
303                     return t;
304             }
305
306             public static void appendCode(String str) {
307                     cppCode.append(str);
308             }
309
310             public static void appendCode(char chr) {
311                     cppCode.append(chr);
312             }
313
314             public static String getCode() {
315                     return cppCode.toString();
316             }
317
318             public static StringBuilder getCodeBuilder() {
319                     return cppCode;
320             }
321
322             public static TypeVariable getExpressionType(String operator,
323                             TypeVariable a, TypeVariable b) throws SymbolException {
324                     int operatorVal = 0;
```

```
325
326
327                 if (a instanceof TypeArray || b instanceof TypeArray)
328                         throw new SymbolException(
329                                         "Expression data type (array) no allowed ");
330
331                 if (operator == "OR" || operator == "AND") {
332                         if (a.matches(TypeBool) && b.matches(TypeBool))
333                                 return TypeBool;
334                         else
335                                 throw new SymbolException("Expression data type not
    allowed ");
336                 }
337                 if (operator == "NOT") {
338                         if (a.matches(TypeBool))
339                                 return TypeBool;
340                         else
341                                 throw new SymbolException("Expression data type not
    allowed ");
342                 }
343                 if (operator == "UNARY") {
344                         if (a.matches(TypeFloat) || a.matches(TypeInt) || a.matches
    (TypeChar))
345                                 return a;
346                         else
347                                 throw new SymbolException("Bad Expression: " +
    operator + " on " +a.getClass() + ". data type not allowed ");
348                 }
349
350                 if (operator == "ARITH")
351                         operatorVal = ARITH;
352                 if (operator == "PLUS")
353                         operatorVal = PLUS;
354                 if (operator == "EQUAL")
355                         operatorVal = EQUAL;
356                 if (operator == "COMPR")
357                         operatorVal = COMPR;
358
359                 TypeVariable t = expressionTable.get(variableIndexTable.get
    (a.getTypeCode())
360                                 + variableIndexTable.get(b.getTypeCode()) +
    operatorVal);
361                 if (t == null)
362                         throw new SymbolException("Expression data type not allowed
    ");
363                 return t;
364         }
365
366         /**
367          * Makes sure the 'main' function exists.
368          * Throws exception is no main present, more than one exists, returns
    anything but an int,
369          *
370          * @return 0 if main exists with no parameters, 1 if it takes string []
371          */
372         public static int checkMain() throws SymbolException{
```

```
373                 List<Type> mains = globalScope.get("main");
374                 if(mains == null || mains.isEmpty())
375                         throw new SymbolException("Function 'main' undeclared");
376                 if(mains.size() > 1)
377                         throw new SymbolException("Function 'main' declared more
    than once. Overloading 'main' not allowed.");
378
379                 Type maybe_main = mains.get(0);
380                 if(!(maybe_main instanceof TypeFunction))
381                         throw new SymbolException("'main' is not declared as a
    function.");
382                 TypeFunction m = (TypeFunction)maybe_main;
383                 if(!m.getReturnType().matches(Env.TypeInt))
384                         throw new SymbolException("Function 'main' must return an
    int");
385                 Vector<TypeParameter> params = m.getParameters();
386                 switch(params.size()){
387                 case 0:
388                         return 0;
389                 case 1:
390                         TypeArray str_array = new TypeArray(TypeString);
391                         TypeParameter p =params.get(0);
392                         if(str_array.matches(p.getType()) && !p.isOptional())
393                                 return 1;
394                 }
395                 throw new SymbolException("Fuction 'main' may either take 0
    parameters or 1 required string[] parameter.");
396         }
397
398         /**
399          * add includes
400          *
401          * @return
402          */
403         public static String generateHeader() {
404                 StringBuilder header = new StringBuilder("#include \"/u/student/
    eh2315/Desktop/tmil/TMIL.h\"\n\n");
405                 return header.toString();
406         }
407
408         public static String generateFooter() throws SymbolException {
409                 int has_main = checkMain();
410                 StringBuilder footer = new StringBuilder();
411                 footer.append("\nint main(int argc, char* argv[]){\n");
412                 if (has_main == 1) {
413                         footer.append(" vector<string> s;\n" + "        for(int i
    =1;i<argc;i++)\n"
414                                         + "             s.push_back(argv[i]);\n" +
    "       return _main(s);\n");
415                 }else{
416                         footer.append("_main();");
417                 }
418                 footer.append("}\n");
419                 return footer.toString();
420         }
421 }
```

```
422
423  /********
424  * File: TMILWalker.g
425  ***********/
426  header {
427          package tmil;
428          import java.util.*;
429          import symbolTable.*;
430          import symbolTable.varTypes.*;
431  }
432
433  class TMILWalker extends TreeParser;
434
435  options {
436          importVocab = TMILantlr;
437          defaultErrorHandler=false;
438  }
439
440  {
441          StringBuilder buffer =  Env.getCodeBuilder();
442
443          // used to keep track of how many loops we entered/exited for break and
     continue statements
444          int loopCount = 0;
445
446          // return type
447          TypeVariable currentReturnType = Env.TypeVoid;
448
449          void error(antlr.collections.AST node, SymbolException e) throws
     RecognitionException {
450                  error(node,e.getMessage());
451          }
452
453          void error(antlr.collections.AST node, String e) throws RecognitionException
     {
454                  StringBuilder err = new StringBuilder();
455                  err.append("Error ");
456                  if(node instanceof TMILTreeNode){
457                          TMILTreeNode n = (TMILTreeNode)node;
458                          err.append(n.toString()+":"+ n.getTColumn());
459                  }else{
460                          err.append(node.getLine()+":"+ node.getColumn());
461                  }
462                  err.append("> "  + e);
463                  error(err.toString());
464          }
465
466          void error(String err) throws RecognitionException{
467                  throw new RecognitionException(err);
468          }
469  }
470
471  // Basic data types
472  // Return: The data type name in string format.
473  rule_type returns [TypeVariable type]
474  {
```

```
475         type = null;
476 }
477 :         "int"                       {type = Env.TypeInt;}
478         | "float"                     {type = Env.TypeFloat;}
479         | "char"                      {type = Env.TypeChar;}
480         | "bool"                      {type = Env.TypeBool;}
481         | "string"                    {type = Env.TypeString;}
482         | "color"                     {type = Env.TypeColor;}
483         | "image"                     {type = Env.TypeImage;}
484         | "coordinate"          {type = Env.TypeCoordinate;}
485         | "text"                      {type = Env.TypeText;}
486         | "void"                      {type = Env.TypeVoid;}
487 ;
488
489 // NOTE: This is used to consume the PROGRAM root node
490 // Return: Nothing useful.
491 program returns [String program]
492 {
493         program = "";
494 }
495 :
496         #(PROGRAM (program2)*)
497         {
498                 program = buffer.toString();
499         }
500 ;
501
502 // NOTE: This is used for selection between DECL and FUNC_DEF.
503 // This rule is needed because you can only use the | operator when you are at the
504 // "top" level... Thanks ANTLR...
505 program2
506 :
507         #(DECL declaration)
508         |
509         #(FUNC_DEF func_def)
510 ;
511
512 // Declaration statement
513 declaration
514 {
515         TypeVariable type;
516 }
517 :
518         type = rule_type
519         {
520                 // type is void in a declaration, cannot allow it
521                 if(type.matches(Env.TypeVoid))
522                 {
523                         error("Detected void type in declaration.");
524                 }
525         }
526
527         #(DECL_LIST (declaration2[type])*)
528 ;
529
530 declaration2[TypeVariable type]
```

```
{
        String identifier;
        Pair array_index, expr;
        TypeVariable new_type = type;
        String assign_code = "";
}
:
        #(DECL_ITEM
                i:IDENTIFIER
                (
                        (expr = expression
                                {
                                        if(!expr.type.canCoerce(new_type))
                                                error(i,"Invalid default assignment for "+ i.getText() +". Types don't match");
                                        assign_code = " = ";

                                        assign_code += expr.cppCode;
                                }
                          )
                        | (#(ARRAY_KEYS array_index = expression
                                {
                                        if(array_index.type.matches(Env.TypeInt)) {
                                                new_type = new TypeArray (new_type,array_index.cppCode);
                                        } else {
                                                error(i,"Array size must be an integer or an integer expresion");
                                        }
                                }
                          ))*
             )
                {
                        try{
                                identifier = i.getText();

                                Env.putVariable(identifier,new_type);

                        // Generate the name into the buffer
                                buffer.append(new_type.getTypeCode());
                                buffer.append(' ');
                                buffer.append(identifier);

                                if(new_type instanceof TypeArray)
                                        buffer.append(((TypeArray) new_type).getInitCode());
                                else
                                        buffer.append(assign_code);

                                buffer.append(";\n");
                        } catch(SymbolException e) {
                                error(i,e);
                        }
                }
        )
;
```

```
583
584   // Function definition
585   func_def
586   {
587           TypeVariable returnType;
588           String funcName, temp;
589           TypeParameter param;
590
591           // Enter scope
592           Env.enterScope();
593   }
594   :
595           returnType = rule_type
596           {
597                   currentReturnType = returnType;
598           }
599           (#(ARRAY_KEYS
600                   {
601                           if(returnType instanceof TypeVoid){
602                                   error(#ARRAY_KEYS,"Void arrays aren't
      allowed");
603                           }
604                           returnType = new TypeArray(returnType);
605                   }
606           ))*
607
608           {
609                           buffer.append(returnType.getTypeCode());
610           }
611
612           i:IDENTIFIER
613           {
614                           funcName = i.getText();
615
616                   TypeFunction function = new TypeFunction(returnType);
617                   buffer.append(' ');
618                   if(funcName.equals("main"))
619                           buffer.append("_");
620                   buffer.append(funcName);
621                   buffer.append("( ");
622           }
623           #(FUNC_DEF_ARG_LIST
624                   (param = func_def_decl[funcName]
625                   {
626                           try{
627                                   function.addParameter(param);
628                                   Env.putVariable(param.getName(), param.getType());
629                           } catch (SymbolException excep){
630                                   error(i,excep);
631                           }
632                   }
633           )*
634           )
635           {
636                   // very lame - needed to remove the last comma in func names
637                   buffer.deleteCharAt(buffer.length() - 1);
```

```
638                     buffer.append(")\n{\n");
639
640                     // add function to global scope.
641                     try{
642                             Env.putFunction(i.getText(), function);
643                     } catch (SymbolException excep) {
644                             error(i, excep);
645                     }
646             }
647             func_body
648             {
649                     buffer.append("}\n");
650             }
651     ;
652
653     func_def_decl[String funcName] returns [TypeParameter param]
654     {
655             TypeVariable type;
656             Pair e;
657             String assign_code = "";
658             param = null;
659     }
660     :
661             #(DECL
662                     type = rule_type
663                     i:IDENTIFIER
664                     (
665                             (e = expression
666                                     {
667                                             if(!e.type.canCoerce(type))
668                                                     error(i,"Invalid default parameter
    assignment for "+ i.getText() +". Types don't match");
669                                             assign_code = "=" + e.cppCode;
670                                     }
671                               )
672                             | (#(ARRAY_KEYS
673                                     {
674                                     type = new TypeArray(type);
675                                     }
676                               ))*
677                 )
678                     {
679                             buffer.append(type.getTypeCode());
680                             buffer.append(' ');
681                             buffer.append(i.getText());
682                             buffer.append(assign_code);
683                             buffer.append(',');
684                             param = new TypeParameter(type, assign_code.length()>0,
    i.getText());
685                     }
686              )
687     ;
688
689     func_body
690     {
691     }
```

```
692   :
693           #(BLOCK (statement["null"])*
694           {
695                   Env.leaveScope();
696           }
697           )
698   ;
699
700   statement[String prepend]
701   {
702           // hack to handle dangling else
703           if(prepend == "else")
704                   buffer.append("else\n");
705   }
706   :
707           #(STATEMENT
708                   (statement2)*
709           )
710           |
711           #(BLOCK
712                   {
713                           Env.enterScope();
714                           buffer.append("{\n");
715                   }
716                   (statement["null"])*
717           )
718           {
719                   Env.leaveScope();
720                   buffer.append("}\n");
721           }
722   ;
723
724   statement2
725   {
726           Pair expr = new Pair();
727           expr.type = Env.TypeVoid;
728   }
729   :
730           #(NULL_NODE {})
731           |
732           #(DECL declaration)
733           {
734                   buffer.append("\n");
735           }
736           |
737           #(STAMP assignment_stamp)
738           {
739                   buffer.append(";\n");
740           }
741           |
742           #("if" if_statement)
743           |
744           #("while" while_statement)
745           |
746           #("for" for_statement)
747           |
```

```
748           /*#(FUNC_CALL expr = func_call)
749           {
750                   buffer.append(expr.cppCode);
751                   buffer.append(";\n");
752           }
753           |*/
754           #("break"
755             {
756                   if(loopCount == 0)
757                   {
758                           error("Break statement needs to be within a loop.");
759                   }
760                   else
761                   {
762                           buffer.append("break;\n");
763                   }
764             }
765            )
766           |
767           #("continue"
768             {
769                   if(loopCount == 0)
770                   {
771                           error("Continue statement needs to be within a loop.");
772                   }
773                   else
774                   {
775                           buffer.append("continue;\n");
776                   }
777             }
778            )
779           |
780           #("return" (expr = expression)?
781           {
782                   buffer.append("return ");
783                   // TODO: These lines probably can be rewritten, but it works so far
784                   if((currentReturnType == Env.TypeVoid) && (expr.type ==
      Env.TypeVoid))
785                   {
786                   }
787                   else if((currentReturnType != Env.TypeVoid) && (expr.type ==
      currentReturnType))
788                   {
789                   }
790                   else if((currentReturnType != Env.TypeVoid) && (expr.type !=
      currentReturnType))
791                   {
792                           error("Return types do not match.");
793                   }
794                   else
795                   {
796                           error("Return types do not match.");
797                   }
798
799                   // take care of the case where it is void so it doesn't output null
800                   if(expr.cppCode != "null")
```

```
801                             buffer.append(expr.cppCode);
802
803                     buffer.append(";\n");
804             }
805             )
806             | expr = expression
807                     {
808                             buffer.append(expr.cppCode);
809                             buffer.append(";\n");
810                     }
811     ;
812
813     assignment_stamp
814     {
815             TypeVariable type;
816             Pair lval = new Pair();
817             Pair expr = new Pair();
818     }
819     :
820             #(L_VALUE lval = l_value[false])
821             {
822                     if(!lval.type.matches(Env.TypeImage))
823                     {
824                             error("Left side of stamp must be an image type.");
825                     }
826             }
827
828             expr = expression
829             {
830                     if(!expr.type.matches(Env.TypeText))
831                     {
832                             error("Right side of stamp must be a text type.");
833                     }
834                     buffer.append("stamp(" + lval.cppCode + "," + expr.cppCode + ")");
835             }
836
837     ;
838
839     l_value[boolean is_assignment] returns [Pair rtn]
840     {
841             rtn = new Pair();
842             Pair array_index;
843             TypeVariable new_type = Env.TypeVoid;
844             StringBuilder return_code = new StringBuilder();
845     }
846     :
847             i:IDENTIFIER
848             {
849                     try{
850                             new_type = Env.getVariable(i.getText());
851                     }catch (SymbolException e) {
852                             error(i,e);
853                     }
854                     return_code.append(i.getText());
855             }
856
```

```
857          (#(ARRAY_KEYS array_index = expression
858            {
859                  if(new_type instanceof TypeArray){
860                          TypeArray type_array = (TypeArray)new_type;
861                          if(array_index.type.matches(Env.TypeInt)) {
862                                  new_type = type_array.getInternalType();
863                                  return_code.append('[');
864                                  return_code.append(array_index.cppCode);
865                                  return_code.append(']');
866                          } else {
867                                  error(i,"Array size must be an integer or an integer
     expresion" );
868                          }
869                  }else{
870                          error(#ARRAY_KEYS,i.getText() + " is not an array.");
871                  }
872            }
873          ))*
874
875          //check for properties
876          (#(PROP_NAMES
877                          prop:IDENTIFIER
878                          {
879                                  try{
880                                          if(new_type instanceof TypeClass){
881                                                  TypeProperty p = ((TypeClass)
     new_type).getProperty(prop.getText());
882                                                  if(is_assignment && p.isReadOnly()){
883                                                          throw new SymbolException
     (prop.getText() + " is a read only property");
884                                                  }
885                                                  return_code.append('.');
886                                                  return_code.append(p.getCppName());
887                                                  new_type = p.getType();
888                                          }else{
889                                                  throw new SymbolException("Cannot
     access propery of non-class type variable");
890                                          }
891                                  }catch(SymbolException se){
892                                          error(prop,se);
893                                  }
894                          }
895
896                  )
897          )*
898          {
899                  rtn.type = new_type;
900                  rtn.cppCode = return_code.toString();
901          }
902
903   ;
904
905   asmt_expr returns [Pair rtn]
906   {
907          rtn = new Pair();
908          StringBuilder cpp_builder = new StringBuilder();
```

```
909             Pair lval;
910             Pair expr;
911     }
912     :#(ASMT
913             #(L_VALUE lval = l_value[true])
914             {
915                     cpp_builder.append(lval.cppCode);
916                     cpp_builder.append(" = ");
917             }
918
919             expr = expression
920             {
921                     if(!expr.type.canCoerce(lval.type))
922                     {
923                             error("Cannot coerce type:" + lval.type.getClass() + " into
    " + expr.type.getClass());
924                     }
925             cpp_builder.append(expr.cppCode);
926
927             rtn.type = lval.type;
928             rtn.cppCode = cpp_builder.toString();
929             }
930     )
931     ;
932
933     if_statement
934     {
935             Pair expr = new Pair();
936     }
937     :
938     {
939             buffer.append("if(");
940             Env.enterScope();
941     }       // NOTE: We do not need to create scopes here because we handle it in
    statement
942             expr = expression
943
944             {
945                     buffer.append(expr.cppCode);
946
947                     if(expr.type.canCoerce(Env.TypeBool))
948                     {
949                             buffer.append(")\n");
950                             Env.enterScope();
951                     }
952                     else
953                     {
954                             error("Expression cannot be coerced into bool.");
955                     }
956             }
957             // the if portion
958             statement["null"]
959
960             {
961                     Env.leaveScope();
962                     // we enter the scope for the else if it exists or if it doesn't
```

```
963                     // if it doesn't, we'll destroy anyway at the end
964                     Env.enterScope();
965             }
966
967             // the else portion
968             (statement["else"])?
969             {
970                     Env.leaveScope();
971             }
972     ;
973
974     while_statement
975     {
976             Pair expr = new Pair();
977
978             loopCount++;
979             buffer.append("while(");
980     }
981     :
982             expr = expression
983             {
984                     buffer.append(expr.cppCode);
985
986                     if(expr.type.canCoerce(Env.TypeBool))
987                     {
988                             buffer.append(")\n");
989                             Env.enterScope();
990                     }
991                     else
992                     {
993                             error("Expression cannot be coerced into bool.");
994                     }
995             }
996
997             // NOTE: We do not need to create scopes here because we handle it in
    statement
998             statement["null"]
999             {
1000                    Env.leaveScope();
1001                    loopCount--;
1002            }
1003    ;
1004
1005
1006    for_statement
1007    {
1008            buffer.append("{\n");
1009            Env.enterScope();
1010            loopCount++;
1011            Pair incr;
1012    }
1013    :
1014            loop_init_cond
1015            { buffer.append(";\nwhile("); }
1016            loop_init_cond
1017            { buffer.append("){\n"); }
```

```
1018
1019            //hold the incrememnt code for a bit
1020            incr = loop_incr
1021
1022
1023            statement["null"]
1024
1025            {
1026                    buffer.append("\n");
1027                    buffer.append(incr.cppCode);
1028                    loopCount--;
1029                    Env.leaveScope();
1030                    buffer.append(";\n}\n}");
1031            }
1032    ;
1033
1034    loop_init_cond
1035    {
1036            Pair expr = new Pair();
1037    }
1038    :
1039             #(NULL_NODE {})
1040            |#(DECL declaration)
1041            | expr = expression
1042            {
1043                    buffer.append(expr.cppCode);
1044            }
1045    ;
1046
1047
1048    loop_incr returns [Pair rtn]
1049    {
1050            rtn = new Pair();
1051    }
1052    : rtn = expression
1053
1054
1055
1056    ;
1057
1058    func_call returns [Pair rtn]
1059    {
1060            String funcName;
1061            java.util.Vector<Pair> temp;
1062            TypeFunction function = null;
1063            rtn = new Pair();
1064    }
1065    :
1066            i:IDENTIFIER
1067            temp = func_arg_list
1068
1069            {
1070                    funcName = i.getText();
1071                    if(funcName == "main")
1072                            funcName = "_"+funcName;
1073                    StringBuilder code = new StringBuilder(funcName+'(');
```

```
1074                    int length = code.length();
1075                    Vector<TypeVariable> parameters = new Vector<TypeVariable>(temp.size
        ());
1076
1077                    for(Pair p : temp){
1078                            parameters.add(p.type);
1079                            code.append(p.cppCode);
1080                            code.append(',');
1081                    }
1082
1083                    // if no arguments supplied, let's not delete the (
1084                    if(length != code.length())
1085                            code.deleteCharAt(code.length()-1);
1086
1087                    code.append(')');
1088
1089                    try{
1090                            function = Env.getFunction(funcName, parameters);
1091                    } catch (SymbolException e){
1092                            error(i,e);
1093                    }
1094
1095                    rtn.type = function.getReturnType();
1096                    rtn.cppCode = code.toString();
1097                    //buffer.append(rtn.cppCode);
1098            }
1099    ;
1100
1101    func_arg_list returns [Vector<Pair> params]
1102    {
1103            Pair expr;
1104            params = new Vector<Pair>();
1105    }
1106    :
1107            #(FUNC_CALL_ARG_LIST
1108                    (expr = expression
1109                             {
1110                                     params.add(expr);
1111                             })*
1112            )
1113    ;
1114
1115    expression returns [Pair command]
1116    {
1117            Pair a = new Pair();
1118            Pair b = new Pair();
1119            command = new Pair();
1120            command.cppCode = "";
1121    }
1122      : #(OR a=expression b=expression
1123         {
1124                    try{command.type = Env.getExpressionType("OR", a.type, b.type);}
1125                    catch (SymbolException e){      error(#OR,e);   }
1126            command.cppCode = "(" + a.cppCode + " || " + b.cppCode + ")";
1127         })
1128      | #(AND a=expression b=expression
```

```
1129        {
1130                    try{command.type = Env.getExpressionType("AND", a.type, b.type);}
1131                    catch (SymbolException e){      error(#AND,e);  }
1132           command.cppCode = "(" + a.cppCode + " && " + b.cppCode + ")";
1133        })
1134     | #(NOT a=expression
1135        {
1136                    try{command.type = Env.getExpressionType("NOT", a.type, a.type);}
1137                    catch (SymbolException e){      error(#NOT,e);  }
1138           command.cppCode = "(" + "!" + a.cppCode + ")";
1139        })
1140     | #(EQUALEQUAL a=expression b=expression
1141        {
1142                    try{command.type = Env.getExpressionType("EQUAL", a.type, b.type);}
1143                    catch (SymbolException e){      error(#EQUALEQUAL,e);   }
1144           command.cppCode = "(" + a.cppCode + " == " +b.cppCode + ")";
1145        })
1146     | #(NOTEQUAL a=expression b=expression
1147        {
1148                    try{command.type = Env.getExpressionType("EQUAL", a.type, b.type);}
1149                    catch (SymbolException e){      error(#NOTEQUAL,e);     }
1150           command.cppCode = "(" + a.cppCode + " != " +b.cppCode + ")";
1151        })
1152     | #(GREATER a=expression b=expression
1153        {
1154                    try{command.type = Env.getExpressionType("COMPR", a.type, b.type);}
1155                    catch (SymbolException e){      error(#GREATER,e);      }
1156           command.cppCode = "(" + a.cppCode + " > " + b.cppCode + ")";
1157        })
1158     | #(GE a=expression b=expression
1159        {
1160                    try{command.type = Env.getExpressionType("COMPR", a.type, b.type);}
1161                    catch (SymbolException e){      error(#GE,e);   }
1162           command.cppCode = "(" + a.cppCode + " >= "+ b.cppCode + ")";
1163        })
1164     | #(LESS a=expression b=expression
1165        {
1166                    try{command.type = Env.getExpressionType("COMPR", a.type, b.type);}
1167                    catch (SymbolException e){      error(#LESS,e); }
1168           command.cppCode = "(" + a.cppCode + " < " + b.cppCode + ")";
1169        })
1170     | #(LE a=expression b=expression
1171        {
1172                    try{command.type = Env.getExpressionType("COMPR", a.type, b.type);}
1173                    catch (SymbolException e){      error(#LE,e);   }
1174           command.cppCode = "(" + a.cppCode + " <= "+ b.cppCode + ")";
1175        })
1176     | #(PLUS a=expression b=expression
1177        {
1178                    try{command.type = Env.getExpressionType("PLUS", a.type, b.type);}
1179                    catch (SymbolException e){      error(#PLUS,e); }
1180           command.cppCode = "(" + a.cppCode + " + " + b.cppCode + ")";
1181        })
1182     | #(MINUS a=expression b=expression
1183        {
1184                    try{command.type = Env.getExpressionType("ARITH", a.type, b.type);}
```

```
1185                    catch (SymbolException e){      error(#MINUS,e);        }
1186            command.cppCode = "(" + a.cppCode + " - " + b.cppCode + ")";
1187         })
1188      | #(MULT a=expression b=expression
1189         {
1190                    try{command.type = Env.getExpressionType("ARITH", a.type, b.type);}
1191                    catch (SymbolException e){      error(#MULT,e); }
1192            command.cppCode = a.cppCode + " * " + b.cppCode;
1193         })
1194      | #(DIV a=expression b=expression
1195         {
1196                    try{command.type = Env.getExpressionType("ARITH", a.type, b.type);}
1197                    catch (SymbolException e){      error(#DIV,e);  }
1198            command.cppCode = a.cppCode + " / " + b.cppCode;
1199         })
1200      | #(MOD a=expression b=expression
1201         {
1202                    try{command.type = Env.getExpressionType("ARITH", a.type, b.type);}
1203                    catch (SymbolException e){      error(#MOD, e); }
1204            command.cppCode = a.cppCode + " % " + b.cppCode;
1205         })
1206      | #(UNI_BEFORE a=uni_before
1207              {
1208                    command.cppCode = a.cppCode;    command.type = a.type;
1209              })
1210      | #(UNI_AFTER a=uni_after
1211              {
1212                    command.cppCode = a.cppCode;    command.type = a.type;
1213              })
1214      | #(L_VALUE a = l_value[false]
1215            {
1216                    command.cppCode = a.cppCode;    command.type = a.type;
1217            }
1218      )
1219      | i:INTEGER { command.type = Env.TypeInt; command.cppCode = i.getText();}
1220      | j:FLOAT_POINT { command.type = Env.TypeFloat; command.cppCode = j.getText();}
1221      | k:CHAR { command.type = Env.TypeChar; command.cppCode = '\''+ k.getText() +
    '\'';}
1222      | l:STRING { command.type = Env.TypeString; command.cppCode = '"'+l.getText()+'"';}
1223      | "true" { command.type = Env.TypeBool; command.cppCode = "1";}
1224      | "false" { command.type = Env.TypeBool; command.cppCode = "0";}
1225      | "null" { command.type = Env.TypeVoid; command.cppCode = "null";}
1226      | command = asmt_expr
1227      | #(FUNC_CALL a = func_call { command.type = a.type; command.cppCode = a.cppCode;
1228        } )
1229    ;
1230
1231    uni_before returns [Pair command]
1232    {
1233            Pair a = new Pair();
1234            command = new Pair();
1235            command.cppCode = "";
1236    }
1237     : #(DECREMENT #(L_VALUE a=l_value[true]
1238            {
1239                    try{command.type = Env.getExpressionType("UNARY", a.type, a.type);}
```

```
1240                    catch (SymbolException e){      error(#DECREMENT,e);    }
1241                    command.cppCode = "--" + a.cppCode;
1242            }))
1243     | #(INCREMENT #(L_VALUE a=l_value[true]
1244            {
1245                    try{command.type = Env.getExpressionType("UNARY", a.type, a.type);}
1246                    catch (SymbolException e){      error(#INCREMENT,e);    }
1247                    command.cppCode = "++" + a.cppCode;
1248            }))
1249     | #(PLUS
1250                    a=expression
1251            {
1252                    try{
1253                            command.type = Env.getExpressionType("UNARY", a.type,
     a.type);
1254                    }
1255                    catch (SymbolException e){
1256                            error(#PLUS,e);
1257                    }
1258                    command.cppCode = "+" + a.cppCode;
1259            })
1260     | #(MINUS a=expression
1261            {
1262                    try{
1263                            command.type = Env.getExpressionType("UNARY", a.type,
     a.type);
1264                    }
1265                    catch (SymbolException e){
1266                            error(#MINUS,e);
1267                    }
1268                    command.cppCode = "-" + a.cppCode;
1269            })
1270     ;
1271     uni_after returns [Pair command]
1272     {
1273            Pair a = new Pair();
1274            command = new Pair();
1275            command.cppCode = "";
1276     }
1277      : #(DECREMENT #(L_VALUE a=l_value[true]
1278            {
1279                    try{command.type = Env.getExpressionType("UNARY", a.type, a.type);}
1280                    catch (SymbolException e){      error(#DECREMENT,e);    }
1281                    command.cppCode = a.cppCode + "--";
1282            }))
1283     | #(INCREMENT #(L_VALUE a=l_value[true]
1284            {
1285                    try{command.type = Env.getExpressionType("UNARY", a.type, a.type);}
1286                    catch (SymbolException e){      error(#INCREMENT,e);    }
1287                    command.cppCode = a.cppCode + "++";
1288            }))
1289     ;
1290
1291     /********
1292     * File: Pair.java
1293     ***********/
```

```
1294  package tmil;
1295
1296  import symbolTable.TypeVariable;
1297
1298
1299  public class Pair {
1300          public TypeVariable type;
1301          public String cppCode;
1302  }
1303
1304  /********
1305  * File: TMILTreeNode.java
1306  ***********/
1307  package tmil;
1308
1309  import antlr.*;
1310  import antlr.collections.*;
1311
1312  public class TMILTreeNode extends CommonAST {
1313          private static final long serialVersionUID = 1L;
1314          private int linet, columnt;
1315
1316      public void initialize(Token t)
1317      {
1318          super.initialize(t);
1319          //System.out.println(t.getColumn());
1320          linet = t.getLine();
1321          columnt = t.getColumn();
1322      }
1323
1324      @Override
1325          public void initialize(int t, String txt) {
1326          //System.out.println(1);
1327                  super.initialize(t, txt);
1328          }
1329
1330          public void initialize(AST t)
1331      {
1332          super.initialize(t);
1333          System.out.println(1);
1334          linet = t.getLine();
1335          columnt = t.getColumn();
1336      }
1337
1338      public int getTLine(){
1339          return linet;
1340      }
1341
1342      public int getTColumn(){
1343          return columnt;
1344      }
1345
1346      public String toString(){
1347          return super.toString() + linet;
1348      }
1349
```

```
1350  }
1351  /********
1352  * File: TMIL2.g
1353  ***********/
1354  header {
1355  package tmil;
1356  }
1357  class TMILLexer extends Lexer;
1358
1359  options {
1360          k = 2;
1361          charVocabulary = '\3'..'\377';
1362          testLiterals = false;
1363          exportVocab = TMILantlr;
1364          defaultErrorHandler=false;
1365  }
1366
1367  tokens {
1368          INTEGER;
1369  }
1370
1371  OPENBRACE: '{';
1372  CLOSEBRACE: '}';
1373  OPENPARENT: '(';
1374  CLOSEPARENT: ')';
1375  OPENBRACKET: '[';
1376  CLOSEBRACKET: ']';
1377  COMMA: ',';
1378  SEMI: ';';
1379  NOT: '!';
1380  EQUAL: '=';
1381  EQUALEQUAL: "==";
1382  NOTEQUAL: "!=";
1383  GE: ">=";
1384  LE: "<=";
1385  GREATER: '>';
1386  LESS: '<';
1387  PLUS: '+';
1388  INCREMENT: "++";
1389  MINUS: '-';
1390  DECREMENT: "--";
1391  MULT: '*';
1392  DIV: '/';
1393  MOD: '%';
1394  AND: "&&";
1395  OR: "||";
1396  STAMP: "<~";
1397
1398  protected DOT: '.';
1399  protected CR: '\r';
1400  protected NL: '\n';
1401  protected TAB: '\t';
1402
1403  protected DIGIT: '0'..'9';
1404  protected LETTER: 'a'..'z' | 'A'..'Z';
1405  protected UNDERSCORE: '_';
```

```
1406
1407  // basically copied from Edwards ANTLR slide 32
1408  COMMENT_MULTI: "/*"
1409                              ( options{greedy = false;}:
1410                                          (
1411                                                  (CR NL) => CR NL { newline();}
1412                                                  | CR {newline();}
1413                                                  | NL {newline();}
1414                                                  | ~('\n' | '\r')
1415                                          )
1416                              )*
1417                              "*/" {$setType(Token.SKIP);};
1418
1419  // single line comments
1420  COMMENT_SINGLE: "//"
1421                                   (options{greedy = false;}:
1422                                          ~('\n' | '\r')
1423                                   )*
1424                                   (
1425                                          (CR NL) => CR NL
1426                                          | CR
1427                                          | NL
1428                                   )      {newline();} {$setType(Token.SKIP);};
1429
1430  // we don't care about the newline
1431  NEWLINE: (
1432                           (CR NL) => CR NL
1433                           | CR
1434                           | NL
1435                  )      {newline();} {$setType(Token.SKIP);};
1436
1437  // we don't care about whitespace
1438  WHITESPACE: (' '| '\t')+ {$setType(Token.SKIP);};
1439
1440  IDENTIFIER options {testLiterals = true;} : LETTER (DIGIT | LETTER | UNDERSCORE)*;
1441
1442  protected EXP_PORTION: 'e' ('+' | '-')? (DIGIT)+;
1443
1444  FLOAT_POINT:(DOT (DIGIT)+) => DOT (DIGIT)+ (EXP_PORTION)?
1445                           | DOT {$setType(DOT);}
1446                           | (DIGIT)+
1447                            (
1448                                   (
1449                                   DOT (DIGIT)* (EXP_PORTION)?
1450                                   | EXP_PORTION
1451                           )
1452                                   | {$setType(INTEGER);}
1453                            );
1454
1455  // strings start and end with "
1456  // how to do the escape was adapted from: http://www.doc.ic.ac.uk/lab/secondyear/
      Antlr/lexer.html
1457  STRING: '"'!
1458                  (
1459                           ~('"' | '\\')
1460                           | STR_ESCAPE
```

```
1461                    )*
1462                    '"'!
1463                    ;
1464
1465    protected STR_ESCAPE: '\\'
1466                                                ('"' { $setText("\\\"");}
1467                                                        | 'n' { $setText("\\n");}
1468                                                        | 'r' { $setText("\\r");}
1469                                                        | 't' { $setText("\\t");}
1470                                                        | '\\' { $setText("\\\\");}
1471                                        );
1472
1473    CHAR: '\''! ~('\'') '\''!;
1474
1475
1476    class TMILParser extends Parser;
1477    options {
1478            k = 3;
1479            buildAST = true;
1480            exportVocab = TMILantlr;
1481            defaultErrorHandler=false;
1482    }
1483
1484    tokens {
1485            PROGRAM;
1486            STATEMENT;
1487            FUNC_DEF;
1488            FUNC_DEFS;
1489            FUNC_DEF_ARG_LIST;
1490            FUNC_CALL;
1491            FUNC_CALL_ARG_LIST;
1492            COLLECTION;
1493            ARRAY_KEYS;
1494            PROP_NAMES;
1495            ASMT;
1496            DECL;
1497            DECL_LIST;
1498            DECL_ITEM;
1499            EXPR;
1500            NULL_NODE;
1501            L_VALUE;
1502            UNI_BEFORE;
1503            UNI_AFTER;
1504            BLOCK;
1505            STAMP_ASMT;
1506    }
1507
1508    program: (func_def | declaration_statement)* EOF!
1509                    {#program = #([PROGRAM, "PROGRAM"], #program);};
1510
1511    /* Declarations */
1512    declaration_statement: declaration SEMI!;
1513    declaration: type declaration_vars {#declaration = #([DECL, "DECL"], #declaration);};
1514    declaration_vars: declaration_vars_id (COMMA! declaration_vars_id)*
      {#declaration_vars = #([DECL_LIST,"DECL_LIST"], #declaration_vars);};
1515    declaration_vars_id: IDENTIFIER ((EQUAL! expression)
```

```
1516                                                                    |
    (l_value_keys)* )
1517
    {#declaration_vars_id = #([DECL_ITEM, "DECL_ASMT"], #declaration_vars_id);};
1518
1519  /* Statements section */
1520  statement: ( SEMI! { #statement = #([NULL_NODE, "NULL_STMT"]); } |//allow for blank
    semi's
1521                    (l_value (EQUAL|STAMP) )=> assignment_stamp_statement |
1522                    declaration_statement |
1523                    break_statement |
1524                    continue_statement |
1525                    for_statement |
1526                    if_statement |
1527                    return_statement |
1528                    while_statement |
1529                    (expression SEMI!) //|
1530                    //(INCREMENT^ | DECREMENT^) IDENTIFIER |
1531                    //IDENTIFIER (INCREMENT^ | DECREMENT^) |
1532                    //func_call SEMI!*/
1533                         ){#statement = #([STATEMENT,"SINGLE_STMT"], #statement);}
1534                    |
1535                    // multiple statements need to be enclosed in braces
1536                    (OPENBRACE! (statement)* CLOSEBRACE! {#statement = #([BLOCK,
    "BLOCK"], #statement);});
1537
1538  break_statement: "break" SEMI!;
1539
1540  continue_statement: "continue" SEMI!;
1541
1542  for_statement
1543    : "for"^ loop_cntrl statement
1544    ;
1545
1546  loop_cntrl
1547    : OPENPARENT! loop_init_cond loop_init_cond loop_incr CLOSEPARENT!
1548    ;
1549
1550  loop_init_cond
1551    : SEMI! { #loop_init_cond = #([NULL_NODE, "NULL_FOR_INIT"]); }
1552    | declaration SEMI!
1553    | (l_value EQUAL) => assignment SEMI!
1554    | expression SEMI!
1555    ;
1556
1557  loop_incr
1558    : { #loop_incr = #([NULL_NODE, "NULL_FOR_INCR"]); }
1559    | (l_value EQUAL) => assignment
1560    | expression
1561    ;
1562
1563  //for_statement: "for"^ OPENPARENT! expression SEMI! expression SEMI! expression
    CLOSEBRACE! statement;
1564
1565  // optional else portion
1566  if_statement: "if"^ OPENPARENT! expression CLOSEPARENT! statement (options
```

```
      {greedy=true;}: "else"! statement)?;
1567
1568  // a return statement can return nothing or an expression
1569  return_statement: "return"^ (expression)? SEMI!;
1570
1571  while_statement: "while"^ OPENPARENT! expression CLOSEPARENT! statement;
1572
1573  /* Types and properties section (reserved keywords) */
1574  type: "int"
1575            | "float"
1576            | "char"
1577            | "bool"
1578            | "string"
1579            | "color"
1580            | "image"
1581            | "coordinate"
1582            | "text"
1583            | "void";
1584
1585
1586  /*property: "x"
1587                  | "y"
1588                  | "h"
1589                  | "w"
1590                  | "rotation"
1591                  | "size"
1592                  | "colour"
1593                  | "R"
1594                  | "G"
1595                  | "B"
1596                  | "position"
1597                  | "name"
1598                  | "font"
1599                  | "length";
1600  */
1601
1602  func_def: type (func_arg_l_value_keys)* IDENTIFIER func_def_arg_list func_body
1603                    {#func_def = #([FUNC_DEF, "FUNC_DEF"], #func_def);};
1604
1605  func_def_decl: type IDENTIFIER (
1606                                                              (EQUAL! expression)
1607                                                              | (func_arg_l_value_keys)* )
1608                                    {#func_def_decl = #([DECL, "FUNC_DECL"],
      #func_def_decl);};
1609
1610  func_arg_l_value_keys: (OPENBRACKET! CLOSEBRACKET!) {#func_arg_l_value_keys = #
      ([ARRAY_KEYS, "FUNC_ARRAY_LEVELS"],
      func_arg_l_value_keys);};
1611
1612  func_def_arg_list: OPENPARENT! (func_def_decl (COMMA! func_def_decl)*)? CLOSEPARENT!
1613                                       {#func_def_arg_list = #([FUNC_DEF_ARG_LIST,
      "FUNC_DEF_ARG_LIST"], func_def_arg_list);};
1614
1615  func_body: OPENBRACE! (statement)* CLOSEBRACE!
1616                                            {#func_body = #([BLOCK,"BLOCK"],
      #func_body);};
```

```
1617
1618  func_call: IDENTIFIER func_call_arg_list
1619                    {#func_call = #([FUNC_CALL, "FUNC_CALL"], #func_call);};
1620  //func_call_arg_list: OPENPARENT! ( IDENTIFIER (COMMA! IDENTIFIER)* )? CLOSEPARENT!
1621  func_call_arg_list: OPENPARENT! (expression (COMMA! expression)* )? CLOSEPARENT!
1622                                          {#func_call_arg_list = #
      ([FUNC_CALL_ARG_LIST, "FUNC_CALL_ARG_LIST"], func_call_arg_list);} ;
1623
1624  /* Expressions section */
1625
1626
1627
1628  stamp: l_value (STAMP! expression) {#stamp = #([STAMP, "STAMP_ASMT"], #stamp);};
1629
1630  assignment_stamp_statement: ((l_value EQUAL) => assignment
1631                                                                  |stamp)
1632                                                          SEMI!;
1633
1634  assignment:  l_value EQUAL! asmt_exp {#assignment = #([ASMT, "ASMT"], #assignment);};
1635  asmt_exp: (l_value EQUAL) => l_value EQUAL! asmt_exp {#asmt_exp = #([ASMT, "ASMT"],
      #asmt_exp);}
1636   | expression;
1637
1638  expression: or_expression ;
1639  or_expression: and_expression (OR^ and_expression)*;
1640  and_expression: not_expression (AND^ not_expression)*;
1641  not_expression: (NOT^)? compare_expression;
1642  compare_expression: add_subtract_expression (
1643  ( EQUALEQUAL^
1644
      NOTEQUAL^
1645
      GREATER^
1646
      GE^
1647
      LESS^
1648
      LE^
1649                                                                                    )
1650                                          add_subtract_expression)*;
1651  add_subtract_expression: multiply_divide_expression ((PLUS^ | MINUS^)
      multiply_divide_expression)*;
1652  multiply_divide_expression: uni_expression (( MULT^ | DIV^ | MOD^) uni_expression)*;
1653  uni_expression:
1654                                  ((INCREMENT^
1655                                          | DECREMENT^ ) l_value {#uni_expression = #
      ([UNI_BEFORE, "UNI_BEFORE"], uni_expression);})
1656                                  | ((PLUS^
1657                                          | MINUS^) (l_value | INTEGER | FLOAT_POINT|
      CHAR ) {#uni_expression = #([UNI_BEFORE, "UNI_BEFORE"], uni_expression);})
1658                                  |(l_value (INCREMENT | DECREMENT)) =>
1659                                          l_value (INCREMENT^ | DECREMENT^)
1660                                          {#uni_expression = #([UNI_AFTER,
      "UNI_AFTER"], uni_expression);}
1661                                  | r_value
```

```
1662                                    ;
1663
1664    r_value: l_value
1665                    | INTEGER
1666                    | FLOAT_POINT
1667                    | CHAR                  // To handle single character like char a =
        '9'; Need to check.
1668                    | STRING
1669                    | "true"
1670                    | "false"
1671                    | "null"
1672                    | func_call
1673                    | (OPENPARENT assignment) => OPENPARENT! assignment CLOSEPARENT!
1674                    | OPENPARENT! expression CLOSEPARENT!
1675                    ;
1676
1677    basic_r_values: l_value
1678                    | INTEGER
1679                    | FLOAT_POINT
1680                    | CHAR                  // To handle single character like char a =
        '9'; Need to check.
1681                    | STRING
1682                    | "true"
1683                    | "false"
1684                    | "null";
1685
1686    // needs to handle the case of arrays and properties
1687    // e.g. a[5][6].x.y = 3;
1688    // e.g. b.x = 4;
1689    l_value: IDENTIFIER (l_value_keys)* (l_value_props)* {#l_value = #([L_VALUE,
        "L_VALUE"], l_value);};
1690
1691    l_value_keys: (OPENBRACKET! expression CLOSEBRACKET!) {#l_value_keys = #
        ([ARRAY_KEYS, "ARRAY_KEYS"], l_value_keys);};
1692
1693    l_value_props: (DOT! IDENTIFIER) {#l_value_props = #([PROP_NAMES, "PROP_NAMES"],
        l_value_props);};
1694
1695
1696    /*****************************
1697    * Directory: src/symbolTable
1698    *****************************/
1699
1700
1701    /********
1702    * File: TypeParameter.java
1703    ***********/
1704    /**
1705     *
1706     */
1707    package symbolTable;
1708
1709    public class TypeParameter{
1710            TypeVariable type;
1711            boolean optional;
1712            String name;
```

```
1713
1714            public TypeParameter(TypeVariable t, boolean optional, String name){
1715                    this.type = t;
1716                    this.optional = optional;
1717                    this.name = name;
1718            }
1719
1720            public String getName(){
1721                    return name;
1722            }
1723
1724            public TypeVariable getType(){
1725                    return type;
1726            }
1727
1728            public boolean isOptional(){
1729                    return optional;
1730            }
1731    }
1732    /********
1733    * File: TypeFunction.java
1734    ***********/
1735    package symbolTable;
1736
1737    import java.util.Vector;
1738
1739    public class TypeFunction extends Type{
1740            private TypeVariable returnType;
1741            private Vector<TypeParameter> parameters;
1742            private int numRequiredParams, numOptionalParams;
1743
1744            public TypeFunction(TypeVariable return_type){
1745                    parameters = new Vector<TypeParameter>();
1746                    this.returnType = return_type;
1747                    numRequiredParams = numOptionalParams = 0;
1748            }
1749
1750            /**
1751             * adds a parameter to the end of the parameter list.
1752             * Warning: once a parameter with a default value has been added (an
        optional parameter),
1753             * parameters without defaults (required parameters) can no longer be added.
1754             * @param name
1755             * @param t
1756             * @param has_valid_default
1757             * @throws SymbolException If a required param is added after an optional one
1758             */
1759            public void addParameter(String name, TypeVariable t, boolean
        has_valid_default) throws SymbolException{
1760                    if(!has_valid_default && numOptionalParams > 0){
1761                            throw new SymbolException("Required parameter -" + name +
1762                                                    "- must be defined before optional
        parameters");
1763                    }
1764                    //increment number of params
1765                    if(has_valid_default)
```

```
1766                            numOptionalParams++;
1767                    else
1768                            numRequiredParams++;
1769                    parameters.add(new TypeParameter(t,has_valid_default,name));
1770            }
1771
1772            public void addParameter(TypeParameter p) throws SymbolException{
1773                    addParameter(p.getName(),p.getType(),p.isOptional());
1774            }
1775
1776            public TypeVariable getReturnType(){
1777                    return returnType;
1778            }
1779
1780            public Vector<TypeParameter> getParameters(){
1781                    return parameters;
1782            }
1783
1784            /**
1785             * Check required parameters of each function to make sure they dont match
    exactly
1786             * @param new_func function to compare
1787             * @return true if the two functions won't conflict, false if the required
1788             * parameters are same
1789             */
1790            public boolean allowOverload(TypeFunction new_func){
1791                    if(this.parameters.size() == 0 && new_func.parameters.size()==0)
1792                            return false;
1793
1794                    //if required number of params dont match, it's an allowed override
1795                    //regardless of param type
1796                    if(this.numRequiredParams != new_func.numRequiredParams)
1797                            return true;
1798
1799                    //run through all required param. if any one of them is
1800                    //not compatible, then allowed
1801                    for(int i = 0; i < this.numRequiredParams; i++){
1802                            TypeVariable my_type = this.parameters.get(i).type;
1803                            TypeVariable his_type = new_func.parameters.get(i).type;
1804                            if(!my_type.canCoerce(his_type))
1805                                    return true;
1806                    }
1807                    return false;
1808            }
1809
1810            /**
1811             * Check if a list of parameters match this function. Only
1812             * one function can match bc's of our overload check above
1813             * @param parameters parameters to check
1814             * @return true if the function matches.
1815             */
1816            public boolean checkCompatibility(Vector<TypeVariable> parameters){
1817                    //make sure enough parameters are passed to fill all the required
    spots
1818                    if(parameters.size() < this.numRequiredParams)
1819                            return false;
```

```
1820                    //make sure too many parameters (more that the function takes)
1821                    //aren't passed int
1822                    if(parameters.size() > this.parameters.size())
1823                            return false;
1824
1825                    for(int i = 0; i<parameters.size(); i++){
1826                            TypeVariable my_type = this.parameters.get(i).type;
1827                            TypeVariable his_type = parameters.get(i);
1828                            if(!my_type.matches(his_type) && !his_type.canCoerce
    (my_type))
1829                                    return false;
1830                    }
1831
1832                    return true;
1833            }
1834
1835
1836    }
1837
1838    /********
1839    * File: TypeClass.java
1840    ***********/
1841    package symbolTable;
1842
1843    import java.util.Hashtable;
1844
1845    /**
1846     * This is abstract. It must be extended for use.
1847     * EXTENDED CLASS MUST define a static block to initialize
1848     *
1849     * @author eh2315
1850     *
1851     */
1852    public abstract class TypeClass extends TypeVariable {
1853
1854            protected static Hashtable<String, TypeProperty> members = new
    Hashtable<String, TypeProperty>();
1855            protected static int BracketLevels = 0;
1856
1857            public TypeProperty getProperty(String prop_name) throws SymbolException{
1858                    TypeProperty t = members.get(prop_name);
1859                    if(t == null)
1860                            throw new SymbolException("Invalid property: " + prop_name +
1861                                            " for type "+ this.getClass().getSimpleName
    ());
1862                    return t;
1863            }
1864
1865    }
1866
1867    /********
1868    * File: SymbolException.java
1869    ***********/
1870    package symbolTable;
1871
1872    public class SymbolException extends Exception {
```

```
1873            private static final long serialVersionUID = 8118111446807146334L;
1874
1875            public SymbolException(String error){
1876                    super(error);
1877            }
1878
1879    }
1880
1881    /********
1882    * File: TypeVariable.java
1883    ***********/
1884    package symbolTable;
1885
1886    public abstract class TypeVariable extends Type {
1887            protected static String cppName;
1888
1889            public abstract boolean canCoerce(Type into);
1890
1891            public abstract String getTypeCode();
1892
1893            public boolean matches(TypeVariable other) {
1894                    if (other.getClass().equals(this.getClass()))
1895                            return true;
1896                    return false;
1897            }
1898
1899            protected TypeVariable checkOpForNumerical(String operator,
1900                            TypeVariable other_type) throws SymbolException {
1901
1902                    return null;
1903            }
1904
1905            public boolean allowBinary(int operator) {
1906                    return false;
1907            }
1908    }
1909
1910    /********
1911    * File: TypeProperty.java
1912    ***********/
1913    /**
1914     *
1915     */
1916    package symbolTable;
1917
1918    public class TypeProperty{
1919            private TypeVariable type;
1920            private boolean readOnly;
1921            private String name;
1922
1923            public TypeProperty(TypeVariable type, boolean read_only, String cpp_name){
1924                    this.type = type;
1925                    this.readOnly = read_only;
1926                    this.name = cpp_name;
1927            }
1928
```

```
1929            public TypeVariable getType(){
1930                    return type;
1931            }
1932
1933            public boolean isReadOnly(){
1934                    return readOnly;
1935            }
1936
1937            public String getCppName(){
1938                    return name;
1939            }
1940
1941    }
1942    /********
1943    * File: varTypes
1944    ***********/
1945
1946    /********
1947    * File: TypeArray.java
1948    ***********/
1949    package symbolTable;
1950
1951    import symbolTable.varTypes.TypeInt;
1952
1953    public class TypeArray extends TypeClass {
1954
1955            static {
1956                    members.put("length", new TypeProperty(new TypeInt(), true, "size
        ()"));
1957            }
1958
1959            TypeVariable internalType;
1960            /**
1961             * Size is optional since for certain instances like
1962             * function declaration, certain sizes aren't needed.
1963             * Value of -1 denotes no size specified
1964             */
1965            String size;
1966
1967            public TypeArray(){
1968
1969            }
1970
1971            public TypeArray(TypeVariable t){
1972                    this();
1973                    setInternalType(t);
1974            }
1975
1976            public TypeArray(TypeVariable t, String size_ccp_code){
1977                    this(t);
1978                    this.size = size_ccp_code;
1979            }
1980
1981            public void setInternalType(TypeVariable t){
1982                    internalType = t;
1983            }
```

```
1984
1985            public TypeVariable getInternalType(){
1986                    return internalType;
1987            }
1988
1989            /**
1990             * converts [][][] to vector code.
1991             */
1992            @Override
1993            public String getTypeCode(){
1994                    return "vector<" + internalType.getTypeCode()+" >";
1995            }
1996
1997            /**
1998             * Generate initialization code that will size the array.
1999             * should work for array of any amount of dimensions
2000             * @return the initialization code for a vector -
2001             */
2002            public String getInitCode(){
2003                    StringBuilder code = new StringBuilder("("+size);
2004                    if(internalType instanceof TypeArray){
2005                            TypeArray internal_array = (TypeArray)internalType;
2006                            code.append(',');
2007                            code.append(internal_array.getTypeCode());
2008                            code.append(internal_array.getInitCode());
2009                    }
2010                    code.append(')');
2011                    return code.toString();
2012            }
2013
2014            @Override
2015            public boolean canCoerce(Type into) {
2016                    if(into instanceof TypeVariable)
2017                            return matches((TypeVariable)into);
2018                    return false;
2019            }
2020
2021            @Override
2022            public boolean matches(TypeVariable other){
2023                    if(other instanceof TypeArray ){
2024                            return internalType.matches(((TypeArray)other).internalType);
2025                    }
2026                    return false;
2027
2028            }
2029
2030    }
2031
2032    /********
2033    * File: SymbolTable.java
2034    ***********/
2035    package symbolTable;
2036
2037    import java.util.ArrayList;
2038    import java.util.Hashtable;
2039    import java.util.List;
```

```
2040  import java.util.Vector;
2041
2042  /**
2043   * Actual Symbol table. Should be access from the Env static class
2044   * @author eh2315
2045   *
2046   */
2047  public class SymbolTable {
2048          /**
2049           * link to parent table for lookups
2050           */
2051          private SymbolTable parent;
2052
2053          /**
2054           * Actual table.  Needs to a List<Type> to handle function
2055           * overloading
2056           */
2057          private Hashtable<String, List<Type>> table;
2058
2059          /**
2060           * Constructs a new level of the table
2061           * Needed for entering scope
2062           * @param parent set to {@code null} if top level
2063           */
2064          public SymbolTable(SymbolTable parent){
2065                  this.parent = parent;
2066                  table = new Hashtable<String, List<Type> >();
2067          }
2068
2069          /**
2070           * returns the parent table. Needed for leaving scope
2071           * @return
2072           */
2073          public SymbolTable getParent(){
2074                  return parent;
2075          }
2076
2077          /**
2078           * used internally to query for an identifier up the scope tree
2079           * @param identifier what to look for
2080           * @return a list of type if found, {@code null} if not found
2081           */
2082          private List<Type> lookup(String identifier){
2083                  for(SymbolTable st = this; st != null; st = st.parent){
2084                          List<Type> t = st.table.get(identifier);
2085                          if(t != null)
2086                                  return t;
2087                  }
2088                  return null;
2089          }
2090
2091          public void putVariable(String identifier, Type t) throws SymbolException{
2092                  //check for the variable in any scope.
2093                  List<Type> list = lookup(identifier);
2094
2095                  //make sure the identifier is not a function
```

```
2096                    if(list != null && list.get(0) instanceof TypeFunction){
2097                            throw new SymbolException("Identifier "+ identifier + "
      already defined as a function");
2098                    }
2099
2100                    if(table.containsKey(identifier)){
2101                            throw new SymbolException("Identifier " + identifier + "
      cannot be redeclared within the same scope level");
2102                    }
2103
2104                    list = new ArrayList<Type>(1);
2105                    list.add(t);
2106                    table.put(identifier, list);
2107            }
2108
2109            public TypeVariable getVariable(String identifier) throws SymbolException{
2110                    List<Type> list = lookup(identifier);
2111                    if (list == null)
2112                            throw new SymbolException("Identifier "+ identifier + " not
      declared in current scope");
2113
2114                    Type t = list.get(0);
2115                    if(t instanceof TypeVariable)
2116                            return (TypeVariable)t;
2117                    throw new SymbolException("Identifier " + identifier + " declared as
      a function, not a variable");
2118            }
2119
2120            public void putFunction(String identifier, TypeFunction new_function) throws
      SymbolException{
2121                    //check for the variable in any scope.
2122                    List<Type> list = lookup(identifier);
2123
2124                    //if the function has never been defined, add it
2125                    if(list == null){
2126                            list = new ArrayList<Type>(1);
2127                            list.add(new_function);
2128                            table.put(identifier, list);
2129                    }
2130                    //make sure the identifier has not already been declared as a
      variable
2131                    else if(list.get(0) instanceof TypeVariable){
2132                            throw new SymbolException("Function " + identifier + "
      already defined as a variable");
2133                    }
2134                    //here we know the identifier exists and is a function, so check
2135                    //for overloading compatability (unqueness) and add it to the list
2136                    //if it's good.
2137                    else{
2138                            //iterate over the list of functions, and throw exception
2139                            //if it has the same required parameter signature
2140                            for(Type list_item : list){
2141                                    TypeFunction func = (TypeFunction)list_item;
2142                                    if(!func.allowOverload(new_function))
2143                                            throw new SymbolException("Function " +
      identifier + " cannot be redeclared/overloaded " +
```

```
2144                                                    "since it has compatible
    required parameters as a previous declaration");
2145                            }
2146                            list.add(new_function);
2147                    }
2148            }
2149
2150            public TypeFunction getFunction(String identifier, Vector<TypeVariable>
    passed_parameters) throws SymbolException{
2151                    List<Type> list = lookup(identifier);
2152                    if(list == null){
2153                            throw new SymbolException("Function " + identifier + " not
    declared");
2154                    }
2155                    if(list.get(0) instanceof TypeVariable){
2156                            throw new SymbolException("Function " + identifier + " not
    declared as a function");
2157                    }
2158                    for(Type list_item : list){
2159                            TypeFunction func = (TypeFunction)list_item;
2160                            if(func.checkCompatibility(passed_parameters))
2161                                    return func;
2162                    }
2163                    throw new SymbolException("Function " + identifier + " with given
    params not declared");
2164            }
2165
2166            /**
2167             * Used for main checking by env.
2168             */
2169            public List<Type> get(String identifier){
2170                    return lookup(identifier);
2171            }
2172
2173    }
2174
2175    /********
2176    * File: Type.java
2177    ***********/
2178    package symbolTable;
2179
2180    public abstract class Type {
2181
2182    }
2183
2184
2185    /*****************************
2186    * Directory: src/symbolTable/varTypes
2187    *****************************/
2188
2189
2190    /********
2191    * File: TypeChar.java
2192    ***********/
2193    package symbolTable.varTypes;
2194
```

```
2195  import symbolTable.Type;
2196  import symbolTable.TypeVariable;
2197
2198  public class TypeChar extends TypeVariable {
2199
2200          @Override
2201          public String getTypeCode() {
2202                  return "char";
2203          }
2204
2205          @Override
2206          public boolean canCoerce(Type into) {
2207                  if (into instanceof TypeBool || into instanceof TypeInt
2208                                  || into instanceof TypeFloat || into instanceof
      TypeChar || into instanceof TypeString)
2209                          return true;
2210                  return false;
2211          }
2212
2213  }
2214
2215  /********
2216  * File: TypeCoordinate.java
2217  ***********/
2218  package symbolTable.varTypes;
2219
2220  import symbolTable.Type;
2221  import symbolTable.TypeClass;
2222  import symbolTable.TypeProperty;
2223
2224  public class TypeCoordinate extends TypeClass {
2225
2226          static {
2227                  members.put("x", new TypeProperty(new TypeInt(), false, "x"));
2228                  members.put("y", new TypeProperty(new TypeInt(), false, "y"));
2229          }
2230
2231          @Override
2232          public String getTypeCode() {
2233                  return "coordinate";
2234          }
2235
2236          @Override
2237          public boolean canCoerce(Type into) {
2238                  if (into instanceof TypeCoordinate)
2239                          return true;
2240                  return false;
2241          }
2242
2243  }
2244
2245  /********
2246  * File: TypeBool.java
2247  ***********/
2248  package symbolTable.varTypes;
2249
```

```
2250  import symbolTable.Type;
2251  import symbolTable.TypeVariable;
2252
2253  public class TypeBool extends TypeVariable {
2254          @Override
2255          public boolean canCoerce(Type into) {
2256                  if (into instanceof TypeBool || into instanceof TypeInt
2257                                  || into instanceof TypeFloat || into instanceof
      TypeChar)
2258                          return true;
2259                  return false;
2260          }
2261
2262          @Override
2263          public String getTypeCode() {
2264                  return "bool";
2265          }
2266
2267  }
2268
2269  /********
2270  * File: TypeString.java
2271  ***********/
2272  package symbolTable.varTypes;
2273
2274  import symbolTable.Type;
2275  import symbolTable.TypeClass;
2276  import symbolTable.TypeProperty;
2277
2278  public class TypeString extends TypeClass {
2279
2280          static {
2281                  members.put("length", new TypeProperty(new TypeInt(), true, "size
      ()"));
2282                  BracketLevels = 1;
2283          }
2284
2285          @Override
2286          public String getTypeCode() {
2287                  return "string";
2288          }
2289
2290          @Override
2291          public boolean canCoerce(Type into) {
2292                  if (into instanceof TypeString)
2293                          return true;
2294                  return false;
2295          }
2296
2297  }
2298
2299  /********
2300  * File: TypeInt.java
2301  ***********/
2302  package symbolTable.varTypes;
2303
```

```
2304  import symbolTable.Type;
2305  import symbolTable.TypeVariable;
2306
2307  public class TypeInt extends TypeVariable {
2308          @Override
2309          public String getTypeCode() {
2310                  return "int";
2311          }
2312
2313          @Override
2314          public boolean canCoerce(Type into) {
2315                  if (into instanceof TypeBool || into instanceof TypeInt
2316                                  || into instanceof TypeFloat || into instanceof
      TypeChar)
2317                          return true;
2318                  return false;
2319          }
2320
2321  }
2322
2323  /********
2324  * File: TypeImage.java
2325  ***********/
2326  package symbolTable.varTypes;
2327
2328  import symbolTable.Type;
2329  import symbolTable.TypeClass;
2330  import symbolTable.TypeProperty;
2331
2332  public class TypeImage extends TypeClass {
2333
2334          static {
2335                  members.put("h", new TypeProperty(new TypeInt(), true, "get_h()"));
2336                  members.put("w", new TypeProperty(new TypeInt(), true, "get_w()"));
2337          }
2338
2339          @Override
2340          public String getTypeCode() {
2341                  return "image";
2342          }
2343
2344          @Override
2345          public boolean canCoerce(Type into) {
2346                  if (into instanceof TypeImage)
2347                          return true;
2348                  return false;
2349          }
2350
2351  }
2352
2353  /********
2354  * File: TypeFloat.java
2355  ***********/
2356  package symbolTable.varTypes;
2357
2358  import symbolTable.Type;
```

```
2359  import symbolTable.TypeVariable;
2360
2361  public class TypeFloat extends TypeVariable {
2362
2363          @Override
2364          public String getTypeCode() {
2365                  return "float";
2366          }
2367
2368          @Override
2369          public boolean canCoerce(Type into) {
2370                  if (into instanceof TypeBool || into instanceof TypeInt
2371                                  || into instanceof TypeFloat || into instanceof
      TypeChar)
2372                          return true;
2373                  return false;
2374          }
2375
2376  }
2377
2378  /********
2379  * File: TypeColor.java
2380  ***********/
2381  package symbolTable.varTypes;
2382
2383  import symbolTable.Type;
2384  import symbolTable.TypeClass;
2385  import symbolTable.TypeProperty;
2386
2387  public class TypeColor extends TypeClass {
2388
2389          static{
2390                  members.put("r", new TypeProperty(new TypeInt(),false,"r"));
2391                  members.put("g", new TypeProperty(new TypeInt(),false,"g"));
2392                  members.put("b", new TypeProperty(new TypeInt(),false,"b"));
2393          }
2394
2395          @Override
2396          public String getTypeCode() {
2397                  return "color";
2398          }
2399
2400          @Override
2401          public boolean canCoerce(Type into) {
2402                  if(into instanceof TypeColor)
2403                          return true;
2404                  return false;
2405          }
2406
2407  }
2408
2409  /********
2410  * File: TypeText.java
2411  ***********/
2412  package symbolTable.varTypes;
2413
```

```
2414  import symbolTable.Type;
2415  import symbolTable.TypeClass;
2416  import symbolTable.TypeProperty;
2417
2418  public class TypeText extends TypeClass {
2419
2420          static {
2421                  members.put("name", new TypeProperty(new TypeString(), false,
      "name"));
2422                  members.put("font", new TypeProperty(new TypeString(), false,
      "font"));
2423                  members.put("colour",
2424                                  new TypeProperty(new TypeColor(), false, "colour"));
2425                  members.put("position", new TypeProperty(new TypeCoordinate(), false,
2426                                  "position"));
2427                  members.put("rotation", new TypeProperty(new TypeInt(), false,
2428                                  "rotation"));
2429                  members.put("size", new TypeProperty(new TypeInt(), false, "size"));
2430          }
2431
2432          @Override
2433          public String getTypeCode() {
2434                  return "word";
2435          }
2436
2437          @Override
2438          public boolean canCoerce(Type into) {
2439                  if (into instanceof TypeText)
2440                          return true;
2441                  return false;
2442          }
2443
2444  }
2445
2446  /********
2447  * File: TypeVoid.java
2448  ***********/
2449  package symbolTable.varTypes;
2450
2451  import symbolTable.Type;
2452  import symbolTable.TypeVariable;
2453
2454  public class TypeVoid extends TypeVariable {
2455
2456          @Override
2457          public String getTypeCode() {
2458                  return "void";
2459          }
2460
2461          @Override
2462          public boolean canCoerce(Type into) {
2463                  if (into instanceof TypeVoid)
2464                          return true;
2465                  return false;
2466          }
2467  }
```

```
2468
2469
2470  /*****************************
2471  * Directory: tests/tmilTest
2472  *****************************/
2473
2474
2475  /********
2476  * File: LexerTester.java
2477  ***********/
2478  package tmilTest;
2479
2480  import java.io.BufferedReader;
2481  import java.io.File;
2482  import java.io.FileNotFoundException;
2483  import java.io.FileReader;
2484  import java.io.Reader;
2485
2486  import tmil.TMILLexer;
2487  import antlr.Token;
2488  import antlr.TokenStreamException;
2489
2490  public class LexerTester extends Tester {
2491
2492          public boolean test(File file) {
2493                  Reader reader;
2494                  try {
2495                          reader = new BufferedReader(new FileReader(file));
2496                  } catch (FileNotFoundException e1) {
2497                          System.err.println("Error reading file");
2498                          return false;
2499                  }
2500                  TMILLexer lexer = new TMILLexer(reader);
2501                  Token token;
2502                  try {
2503                          token = lexer.nextToken();
2504                          while (token.getType() != TMILLexer.EOF) {
2505                                  token = lexer.nextToken();
2506                          }
2507                  } catch (TokenStreamException e) {
2508                          return false;
2509                  } catch (Exception e){
2510                          return false;
2511                  }
2512                  return true;
2513          }
2514
2515          public static void main(String args[]){
2516                  Tester t= new LexerTester();
2517                  t.runTest(t.parseArgs(args));
2518          }
2519  }
2520
2521  /********
2522  * File: GccTester.java
2523  ***********/
```

```
2524  package tmilTest;
2525
2526  import java.io.BufferedReader;
2527  import java.io.File;
2528  import java.io.IOException;
2529  import java.io.InputStreamReader;
2530
2531  public class GccTester extends Tester {
2532          private static String gcc_suffix = " -L\"/u/student/eh2315/lib_dir/libft/lib
      \" -L\"~eh2315/lib_dir/libgd\" -lgd";
2533
2534          @Override
2535          public boolean test(File file) {
2536                  try{
2537                          File output = new File("nul");
2538                          output.deleteOnExit();
2539                          StringBuilder err = new StringBuilder("ERROR: G++ OUPUT:\n");
2540                          String gcc_command = "g++ "+file.getCanonicalPath() + " -o
      "+ output.getName() + gcc_suffix;
2541                          System.out.println("Invoking g++...");
2542                          System.out.println(gcc_command);
2543                          /*
2544                           * Code to invoke external process adapted from:
2545                           * http://www.rgagnon.com/javadetails/java-0014.html
2546                           */
2547
2548                          Process p = Runtime.getRuntime().exec(gcc_command);
2549                          String line;
2550                          BufferedReader gcc_output = new BufferedReader(
2551                                          new InputStreamReader(p.getErrorStream()));
2552                          while ((line = gcc_output.readLine()) != null) {
2553                                  err.append("ERROR: " + line + "\n");
2554                          }
2555                          gcc_output.close();
2556
2557                          if(p.waitFor() == 0)
2558                                  return true;
2559                          else{
2560                                  System.err.println(err.toString());
2561                                  return false;
2562                          }
2563                  } catch (IOException e) {
2564                          System.err.println("Error: "+e.getLocalizedMessage());
2565                          return false;
2566                  } catch (InterruptedException e) {
2567                          System.err.println("Error: "+e.getLocalizedMessage());
2568                          return false;
2569                  }
2570
2571          }
2572
2573          public static void main(String args[]){
2574                  Tester t= new GccTester();
2575
2576                  if (args.length != 1 && args.length !=2) {
2577                          System.err.println("This program takes 1 or 2 params: \nthe
```

```
directory to check and optionally the gcc suffix");
2578                            System.exit(1);
2579                    }
2580                    String dir_path = args[0];
2581                    System.out.println("Directory '" + dir_path + "' selected.");
2582                    if(args.length==2){
2583                            gcc_suffix = args[1];
2584                    }
2585                    System.out.println("G++ Suffix: " + gcc_suffix);
2586                    t.runTest(dir_path);
2587            }
2588
2589    }
2590
2591    /********
2592    * File: ParserTester.java
2593    ***********/
2594    package tmilTest;
2595
2596    import java.io.BufferedReader;
2597    import java.io.File;
2598    import java.io.FileNotFoundException;
2599    import java.io.FileReader;
2600    import java.io.Reader;
2601
2602    import tmil.TMILLexer;
2603    import tmil.TMILParser;
2604    import antlr.RecognitionException;
2605    import antlr.TokenStreamException;
2606
2607    public class ParserTester extends Tester {
2608
2609            @Override
2610            public boolean test(File file) {
2611                    Reader reader;
2612                    try {
2613                            reader = new BufferedReader(new FileReader(file));
2614                    } catch (FileNotFoundException e1) {
2615                            System.err.println("Error reading file");
2616                            return false;
2617                    }
2618
2619            TMILLexer lexer = new TMILLexer(reader);
2620            TMILParser parser = new TMILParser(lexer);
2621            try {
2622                    parser.program();
2623                    } catch (RecognitionException e) {
2624                            return false;
2625                    } catch (TokenStreamException e) {
2626                            System.err.print("Token Streamer error " +
    e.getLocalizedMessage());
2627                            return false;
2628                    } catch(Exception e){
2629                            return false;
2630                    }
2631
```

```
2632            return true;
2633            }
2634
2635            public static void main(String args[]){
2636                    Tester t= new ParserTester();
2637                    t.runTest(t.parseArgs(args));
2638            }
2639
2640    }
2641
2642    /********
2643    * File: individualFileTests
2644    ***********/
2645
2646    /********
2647    * File: Tester.java
2648    ***********/
2649    package tmilTest;
2650
2651    import java.io.File;
2652
2653    public abstract class Tester {
2654
2655            int passed = 0;
2656            int failed = 0;
2657            int counter = 1;
2658
2659            public abstract boolean test(File file);
2660
2661            public void runTest(String dir_path) {
2662                    System.out.println("Directory '" + dir_path + "' selected.");
2663                    File directory = new File(dir_path);
2664                    if (!directory.exists() || !directory.isDirectory()) {
2665                            System.err.println("Directory does not exists, or is a
    file");
2666                            return;
2667                    }
2668
2669                    File[] files = directory.listFiles();
2670                    System.out.println(files.length + " entries found. testing...");
2671
2672                    for (File file : files) {
2673                            if (file.exists() && file.isFile()) {
2674                                    boolean result = test(file);
2675                                    String name = file.getName();
2676                                    if(name.contains(".good"))
2677                                            countPrintTestLine(name, true, result);
2678                                    else if(name.contains(".bad"))
2679                                            countPrintTestLine(name, false, result);
2680                                    else{
2681                                            System.out.println("name missing '.good' or
    '.bad'\t"+name);
2682                                    }
2683                            }
2684                    }
2685
```

```
2686                    System.out.println("Summary: " + (counter - 1) + " tests total\t"
2687                                    + passed + " passed\t" + failed + " failed");
2688                    System.out.println("Done testing.");
2689            }
2690
2691            protected String parseArgs(String args[]) {
2692                    if (args.length != 1) {
2693                            System.err
2694                                            .println("This program takes 1 param, the
    directory to check");
2695                            System.exit(1);
2696                    }
2697                    String dir_path = args[0];
2698                    System.out.println("Directory '" + dir_path + "' selected.");
2699                    return dir_path;
2700            }
2701
2702            protected void countPrintTestLine(String filename, boolean expected, boolean
    actual){
2703                    System.out.print((counter++) + ": Testing '");
2704                    System.out.print(filename + "'. Expected: ");
2705                    String result_seperator = "  Actual: ";
2706                    String ok = "\t Test: OK";
2707                    String bad = "\t Test: BAD";
2708                    if(expected){
2709                            System.out.print("pass" + result_seperator);
2710                            if(actual){
2711                                    System.out.println("pass" + ok);
2712                                    passed++;
2713                            }else{
2714                                    System.out.println("FAIL" + bad);
2715                                    failed++;
2716                            }
2717                    }else{
2718                            System.out.print("fail" + result_seperator);
2719                            if(actual){
2720                                    System.out.println("PASS" + bad);
2721                                    failed++;
2722                            }else{
2723                                    System.out.println("fail" + ok);
2724                                    passed++;
2725                            }
2726                    }
2727            }
2728    }
2729
2730    /********
2731    * File: TMILTester.java
2732    ***********/
2733    package tmilTest;
2734
2735    import java.io.BufferedReader;
2736    import java.io.File;
2737    import java.io.FileNotFoundException;
2738    import java.io.FileReader;
2739    import java.io.Reader;
```

```
2740
2741  import tmil.TMILLexer;
2742  import tmil.TMILParser;
2743  import tmil.TMILWalker;
2744  import antlr.CommonAST;
2745  import antlr.RecognitionException;
2746  import antlr.TokenStreamException;
2747
2748  public class TMILTester extends Tester {
2749
2750          private static boolean verbose = false;
2751
2752          private static void printVerbose(String in) {
2753                  if (verbose)
2754                          System.err.println(in);
2755          }
2756
2757          public boolean test(File file) {
2758                  Reader reader;
2759                  try {
2760                          reader = new BufferedReader(new FileReader(file));
2761                          TMILLexer lexer = new TMILLexer(reader);
2762                          TMILParser parser = new TMILParser(lexer);
2763                          parser.program();
2764                          CommonAST t = (CommonAST) parser.getAST();
2765                          TMILWalker walker = new TMILWalker();
2766                          walker.program(t);
2767                          return true;
2768                  } catch (FileNotFoundException e) {
2769                          return false; // wont happen. checked laststep.
2770                  } catch (RecognitionException e) {
2771                          printVerbose(e.getMessage());
2772                          return false;
2773                  } catch (TokenStreamException e) {
2774                          printVerbose(e.getLocalizedMessage());
2775                          return false;
2776                  }
2777          }
2778
2779
2780          public static void main(String args[]) {
2781                  TMILTester t = new TMILTester();
2782                  if (args.length != 1 && args.length != 2) {
2783                          System.err
2784                                                  .println("This program takes 1 or 2 param,
      the directory to check and verbose mode");
2785                          System.exit(1);
2786                  }
2787                  if (args.length == 2 && Boolean.parseBoolean(args[1]))
2788                          verbose = true;
2789                  t.runTest(args[0]);
2790          }
2791  }
2792
2793  /*****************************
2794  * Directory: tests/cpp_tests
```

```
2795  ******************************/
2796
2797
2798  /********
2799  * File: comments.good.cpp
2800  ***********/
2801  #include "tmil.h"
2802
2803  int main() {
2804
2805  int a;
2806
2807  return 0;
2808
2809
2810
2811  }
2812  /********
2813  * File: global_variables.good.cpp
2814  ***********/
2815  #include "tmil.h"
2816
2817
2818  int a=2;
2819
2820  int main(){
2821
2822          int e = 3;
2823
2824          return a+e;
2825
2826  }
2827  /********
2828  * File: nested_expressions.good.cpp
2829  ***********/
2830  #include "tmil.h"
2831
2832
2833  int main(){
2834
2835          bool x, y z;
2836          int r,s,t;
2837
2838          float c;
2839
2840
2841          x = y=z = 1;
2842
2843          c = 3.4;
2844
2845          c = ( (((r+3*s) %t) * (r+4)) /3);
2846
2847          bool a = x||((y&&z)||x);
2848
2849          return 0;
2850
```

```
2851
2852
2853  }
2854  /********
2855  * File: control_flow.good.cpp
2856  ***********/
2857  #include "tmil.h"
2858
2859
2860  int foo(int x) {
2861
2862          return x - 1;
2863  }
2864
2865  int main(){
2866
2867          bool aBool ;
2868          char aChar;
2869          color aColor ;
2870          float aFloat, bFloat ;
2871          coordinate aCoor, bCoor;
2872          string aString, bString;
2873          word aText;
2874          image aImage;
2875          int aInt, bInt, cInt;
2876
2877          vector<word> arr(3);
2878
2879          aInt = 2;
2880
2881          if(aInt) {
2882
2883                  aBool = 0;
2884          }
2885          else {
2886
2887                  if(aInt>3){
2888
2889                          continue;
2890
2891                  }
2892          }
2893
2894          while(aInt>0){
2895
2896                  aInt = foo(aint);
2897
2898                  if(aInt==1){
2899
2900                          break;
2901
2902                  }
2903
2904          }
2905
2906
```

```
2907            for(int x=3;x<5;x++){
2908
2909                    aInt++;
2910
2911            }
2912
2913            for(int y=3;y!=5;y=y+2) {
2914
2915                    aInt++;
2916
2917            }
2918
2919
2920            return 0;
2921
2922    }
2923    /********
2924    * File: operators_1.good.cpp
2925    ***********/
2926    #include "tmil.h"
2927
2928
2929
2930    int main()
2931    {
2932
2933            bool aBool, bBool ;
2934            char aChar, bChar;
2935            color aColor, bColor ;
2936            float aFloat, bFloat ;
2937            coordinate aCoor, bCoor;
2938            string aString, bString;
2939            word aText;
2940            image aImage;
2941            int aInt, bInt, cInt;
2942
2943            vector <word> arr(3);
2944
2945            // assignment
2946
2947            aBool = true;
2948            bBool = aBool;
2949            aBool = bBool = false;
2950
2951            aChar = 'y';
2952            bChar = aChar;
2953
2954            aColor.r = 255;
2955            bColor.g = aColor.b = 3;
2956
2957
2958            aFloat = 7;
2959            bFloat = -3.544;
2960            aFloat = bFloat;
2961
2962            aCoor.x = 37;
```

```
2963            bCoor = aCoor;
2964
2965            aString = "Hi";
2966            aString = "Hello";
2967            bString = aString;
2968
2969
2970            aText.name = "Hi";
2971            aText.name = bString;
2972            aText.font = "Times.ttf";
2973            aText.colour.r = 255;
2974            aText.colour.g = bColor.b;
2975            aText.position.x = 40;
2976            aText.position.x = aText.position.y = 245;
2977            aText.position = aCoor;
2978            aText.rotation = 34;
2979            aText.rotation = aInt;
2980            aText.size = 7;
2981            aText.size = bInt;
2982
2983            aInt = bInt = cInt = 3;
2984
2985            arr[0].name = "Hello";
2986            arr[1].colour = d;
2987
2988            aInt++;
2989            aText.position.x-- ;
2990
2991
2992            aInt = bInt + 7;
2993            bFloat = bFloat - aFloat + 4;
2994
2995            aBool = !bBool;
2996
2997            aInt = -bInt;
2998
2999            afloat = aText.position.y * aText.rotation / 2 % 4;
3000
3001            return 0;
3002
3003
3004    }
3005    /*********
3006    * File: array_access.good.cpp
3007    ************/
3008    #include "tmil.h"
3009
3010    int main()
3011    {
3012
3013            vector<int> e[(10);
3014
3015            vector<image> a(10);
3016
3017            color c;
3018
```

```
3019          c.r = c.g = c.g = 20;
3020
3021          for(int i= 0; i<10; i++){
3022                  e[i] = i;
3023
3024                  create(image[i],e[i],e[i] +20, c);
3025          }
3026
3027          return 0;
3028
3029
3030  }
3031  /********
3032  * File: declarations_plus_assignments.good.cpp
3033  ***********/
3034  #include "tmil.h"
3035
3036  int main()
3037  {
3038
3039          bool a = false;
3040          char b = 'e';
3041          color d ;
3042          float e = 5.8;
3043          coordinate f;
3044          string g = "Hello world";
3045          word h;
3046          image i;
3047          int l,m ,n;
3048
3049          vector<word> o(3);
3050
3051          d.r = 255;
3052
3053          h.name = "Hi";
3054          h.name = g;
3055
3056          h.font = "Times.ttf";
3057
3058          h.colour.r = 255;
3059          h.colour.g = d.b;
3060
3061        f.x = 2;
3062
3063          h.position.x = 40;
3064
3065          h.rotation = 34;
3066
3067          h.size = 7;
3068
3069          o[0].name = "Hello";
3070          o[1].colour = d;
3071
3072
3073          int p = l;
3074
```

```
3075          color d1;
3076
3077          d1.r = d1.g = d1.b = 25;
3078
3079          return 0;
3080
3081  }
3082  /********
3083  * File: declarations.good.cpp
3084  ***********/
3085  #include "tmil.h"
3086
3087
3088  int main()
3089  {
3090
3091          bool a;
3092          char b;
3093          color d;
3094          float e;
3095          coordinate f;
3096          string g;
3097          word h;
3098          image i;
3099          int l,m ,n;
3100
3101          vector<word> o(3);
3102
3103          return 0;
3104
3105
3106  }
3107  /********
3108  * File: operators_2.good.cpp
3109  ***********/
3110  #include "tmil.h"
3111
3112
3113  int main()
3114  {
3115
3116          bool aBool, bBool ;
3117          char aChar, bChar;
3118          color aColor, bColor ;
3119          float aFloat, bFloat ;
3120          coordinate aCoor, bCoor;
3121          string aString, bString = "try.txt";
3122          word aText ;
3123          image aImage;
3124          int aInt, bInt, cInt;
3125
3126          vector <word> arr(3);
3127
3128          aInt = 4;
3129
3130          aCoor.y = 7;
```

```
3131
3132
3133            if(aInt==5) {
3134
3135                    aBool = true;
3136
3137            }
3138
3139
3140            if(aBool) {
3141
3142                    if(aCoor.y != 3) {
3143
3144                            bInt = 3;
3145
3146                    }
3147            }
3148
3149
3150            bBool = (bInt<=3);
3151
3152           aBool = (aInt>5)&&(aInt<10)||3>=2&&aCoor.y<=7;
3153
3154
3155            aImage = open(bString);
3156
3157            aText.name = "Hi";
3158            aText.font = "Arial.ttf";
3159            aText.colour.r = 255;
3160            aText.position.x = 40;
3161            aText.rotation = 34;
3162            aText.size = 7;
3163
3164            aImage <~ t;
3165
3166
3167            return 0;
3168
3169
3170    }
3171    /*********
3172    * File: fun_declaration_plus_overloading.good.cpp
3173    ************/
3174    #include "tmil.h"
3175
3176    int foo(bool a, coordinate b)
3177    {
3178            return b.x;
3179    }
3180
3181    int foo(bool a)
3182    {
3183            return 1;
3184    }
3185
3186
```

```
3187  int main(){
3188
3189
3190          int a;
3191          bool b = true;
3192          coordinate c;
3193
3194          c.x = foo(b);
3195
3196          a = foo(b,c);
3197
3198          return 0;
3199
3200  }
3201
3202
3203  /*****************************
3204  * Directory: tests/full_program_tests
3205  *****************************/
3206
3207
3208  /********
3209  * File: test1.good.txt
3210  ***********/
3211
3212  //***************************** TEST 1 *******************************
3213
3214
3215  void print_captcha(image im, string s, int rot[], int num , color col, string font) {
3216
3217          coordinate p1,p2;
3218
3219          text w1;
3220
3221          w1.font = font;
3222          w1.colour = col;
3223          w1.size = 120;
3224          w1.position.y = 170;
3225
3226          int val = 30;
3227
3228          for(int i=0; i<num; i++ ){
3229
3230                  val = -val;
3231                  w1.name = char_at(s,i);
3232                  w1.position.x = 20 + i*100;
3233                  w1.position.y = w1.position.y + val;
3234                  w1.rotation = rot[i];
3235
3236                  im <~ w1;
3237          }
3238
3239  }
3240
3241
3242  void draw_grid(image im, int intervalHor, int intervalVer, color c){
```

```
3243
3244            coordinate p1, p2;
3245
3246            p1.y = 0;
3247            p2.y = im.h - 1 ;
3248
3249            for(int i=10;i<im.w;i=i+50){
3250                    p1.x = p2.x = i;
3251                    drawline(im, p1,p2,c, 7);
3252            }
3253
3254            p1.x = 0;
3255            p2.x = im.w - 1;
3256
3257            for(int j=40; j<=im.h; j=j+60) {
3258                    p1.y = p2.y = j;
3259                    drawline(im, p1,p2,c, 7);
3260            }
3261    }
3262
3263
3264    int main() {
3265
3266            color c;
3267            color col;
3268
3269            col.r = 255;
3270            col.g = 222;
3271            col.b = 173;
3272
3273            c.r = 150;
3274            c.g = c.b = 0;
3275
3276            image im;
3277
3278            string s = "PA8";
3279
3280            string font = "Arial.ttf";
3281
3282            create(im,200,400, col);
3283
3284            int rot[3];
3285            rot[0] = 4;
3286            rot[1] = 0;
3287            rot[2] = -7;
3288
3289            print_captcha(im, s,  rot, 3 , c, font) ;
3290
3291            draw_grid(im, 60, 40, c);
3292
3293            save(im,"test1.png");
3294
3295            return 0;
3296
3297    }
3298    /********
```

```
3299  * File: test8.good.txt
3300  ***********/
3301  //*********************************  TEST 8 *****************
3302
3303  int main(){
3304
3305          image im;
3306          open(im, "turtle.jpg");
3307
3308          save(im,"./turtle/turtle0.jpg");
3309
3310          string name;
3311
3312
3313          text w1;
3314          w1.name = "turtle";
3315          w1.font = "GOTHIC.ttf";
3316          w1.rotation = 0;
3317          w1.size = 80;
3318          w1.position.x = im.w/2;
3319          w1.position.y = im.h/2;
3320          w1.colour.r = w1.colour.g = w1.colour.b = 255;
3321
3322
3323          for(int j = 0; j<15; j++){
3324
3325                  w1.colour.b = w1.colour.b - 30;
3326                  w1.colour.g = w1.colour.g - 10;
3327
3328                  w1.rotation = -j*(360/15);
3329
3330                  im <~ w1;
3331
3332                  name = "./turtle/turtle" + int2string(j+1) + ".jpg";
3333
3334                  save(im,name);
3335
3336
3337          }
3338
3339          return 0;
3340
3341  }
3342  /********
3343  * File: test7.good.txt
3344  ***********/
3345  //*********************************  TEST 7 *********************
3346
3347  int main(){
3348
3349          image im;
3350          open(im, "monument.jpg");
3351
3352          image imVec[im.h/10 +1];
3353          string names[im.h/10 +1];
3354
```

```
3355          for(int j = 0; j<(im.h/10 +1); j++){
3356
3357                  imVec[j] = im;
3358                  names[j] = "./monument/" + int2string(j) + ".jpg";
3359          }
3360
3361          text w1, w2;
3362          w1.name = "this is";
3363          w1.font = "Times.ttf";
3364          w1.rotation = 0;
3365          w1.size = 100;
3366          w1.position.x = 120;
3367          w1.colour.r = w1.colour.g = w1.colour.b = 255;
3368
3369          w2 = w1;
3370
3371          w2.name = "an animation";
3372          w2.position.x = 60;
3373
3374          int count = 0;
3375
3376          for(int i=im.h + 100; i>=100; i=i-10) {
3377                  w1.position.y = i;
3378                  w2.position.y = i + 100;
3379
3380                  w1.colour.b = w1.colour.b - 255/((im.w+150)/14);
3381                  w2.colour.b = w1.colour.b;
3382
3383                  imVec[count] <~ w1;
3384                  imVec[count] <~ w2;
3385
3386                  save(imVec[count],names[count]);
3387
3388                  count++;
3389
3390          }
3391
3392          return 0;
3393
3394  }
3395
3396  /********
3397  * File: test2.good.txt
3398  ***********/
3399  //**************************** TEST 2 *******************************
3400
3401  int main() {
3402
3403                  int offset = 3;
3404                  color white,  black,gray;
3405
3406                  white.r = white.g = white.b = 255;
3407                  black.r = black.g = black.b = 0;
3408                  gray.r = gray.g = gray.b = 255;
3409
3410                  coordinate p1, p2;
```

```
3411
3412                    p1.x = 100 + offset;
3413                    p1.y = 180 - offset;
3414
3415                    image yahoo;
3416
3417                    text w1;
3418
3419                    w1.name = "HXV";
3420                    w1.font = "Times.ttf";
3421                    w1.colour = gray;
3422                    w1.position = p1;
3423                    w1.rotation = 4;
3424                    w1.size = 120;
3425
3426                    create(yahoo,200,700,white);
3427
3428                    yahoo <~ w1;
3429
3430                    w1.colour = black;
3431                    w1.position.x = w1.position.x + offset;
3432                    w1.position.y = w1.position.y + offset;
3433
3434                    yahoo <~ w1;
3435
3436
3437                    w1.name ="y";
3438                    w1.colour = black;
3439                    w1.position.x = 450;
3440                    w1.position.y = w1.position.y - 30;
3441                    w1.size = 100;
3442                    w1.rotation = -7;
3443
3444                    yahoo <~ w1;
3445
3446                    w1.name = "2";
3447                    w1.position.x = 620;
3448                    w1.size = 110;
3449                    w1.rotation = 50;
3450
3451
3452                    yahoo <~ w1;
3453
3454                    p1.x = 105;
3455                    p1.y = 120;
3456                    p2.x = 315;
3457                    p2.y = 55;
3458
3459                    drawline(yahoo, p1,p2,black, 5);
3460
3461                    p1.x = p2.x + 62;
3462                    p1.y = p2.y + 77;
3463
3464                    drawline(yahoo, p2,p1,black, 7);
3465
3466                    p2.x = p1.x + 145;
```

```
3467                    p2.y = p1.y - 45;
3468
3469                    drawline(yahoo, p1,p2,black, 5);
3470
3471
3472                    p1.x = 460;
3473                    p1.y = 170;
3474                    p2.x = 650;
3475                    p2.y = 100;
3476
3477                    drawline(yahoo, p1,p2,black, 6);
3478
3479
3480                    save(yahoo,"test2.png");
3481
3482                    return 0;
3483
3484    }
3485    /********
3486    * File: test6.good.txt
3487    ***********/
3488    //*********************************  TEST 6 **********************
3489
3490    void draw_star(image im, color c, int thickness, int numVerteces){
3491
3492            coordinate p[numVerteces];
3493            int i;
3494
3495            p[0].x = im.w/2;
3496            p[0].y = im.h/5;
3497            p[1].x = im.w/5;
3498            p[1].y = im.h/3;
3499            p[2].x = im.w/3;
3500            p[2].y = im.h - im.h/3;
3501            p[3].x = im.w - im.w/3;
3502            p[3].y = im.h - im.h/3;
3503            p[4].x = im.w - im.w/5;
3504            p[4].y = im.h/3;
3505
3506
3507            for(i=0; i<numVerteces; i++){  //
3508
3509                    drawline(im, p[i], p[(i+2)%numVerteces], c, thickness);
3510                    drawline(im, p[i], p[(i+3)%numVerteces], c, thickness);
3511            }
3512
3513    }
3514
3515    int main() {
3516
3517            image im;
3518
3519            color c1;
3520            color c2;
3521
3522            c1.r = c1.g = c1.b = 0;
```

```
3523          c2.r = c2.g = c2.b = 255;
3524
3525          create(im,400,400,c2);
3526          draw_star(im, c1, 5, 5);
3527
3528          save(im, "test6.png");
3529
3530          return 0;
3531
3532
3533  }
3534  /********
3535  * File: test4.good.txt
3536  ***********/
3537  //*************************** TEST 4 ***************************
3538
3539  void zigzag2(image im, color c, int interval){
3540
3541          coordinate p1,p2;
3542
3543          int i;
3544
3545          p1.x = p2.y = 1;
3546
3547          p1.y = interval;
3548          p2.x = im.w -1 ;
3549
3550          for(i=0;i<im.h/interval;i++){
3551
3552                  if(i%2>0)
3553                          p2.y = p1.y + interval;
3554                  else
3555                          p1.y = p2.y + interval;
3556
3557                  drawline(im,p1,p2,c,3);
3558
3559          }
3560
3561          p1.x = 10;
3562          p1.y = im.w - 1;
3563
3564          p2.x = p1.x + interval;
3565          p2.y = 1;
3566
3567          for(i=0;i<im.w/interval;i++){
3568
3569                  if(i%2>0)
3570                          p1.x = p2.x + interval;
3571                  else
3572                          p2.x = p1.x + interval;
3573
3574                  drawline(im,p1,p2,c,3);
3575
3576          }
3577
3578  }
```

```
3579
3580
3581  int main(){
3582
3583          image im1;
3584
3585          open(im1,"Dock.jpg");
3586
3587          color red;
3588          red.r = 255;
3589          red.g = red.b = 0;
3590
3591          coordinate p1;
3592          p1.x = 100;
3593          p1.y = 150;
3594
3595          text w1;
3596
3597          w1.name = "tmil";
3598          w1.font = "CURLZ___.ttf";
3599          w1.colour = red;
3600          w1.position = p1;
3601          w1.rotation = -12;
3602          w1.size = 150;
3603          im1 <~ w1;
3604
3605          zigzag2(im1, red, 60);
3606
3607          save(im1,"test4.jpg");
3608
3609          return 0;
3610
3611  }
3612  /********
3613  * File: test5.good.txt
3614  ***********/
3615  //**************************** TEST 5 ****************************
3616
3617
3618  int main(){
3619
3620          int i = 0;
3621
3622          image im;
3623
3624          open(im,"Dock.jpg");
3625
3626          text t;
3627
3628          t.name = "dog";
3629
3630          t.font = "Arial.ttf";
3631          t.size = 60;
3632
3633          coordinate coor[3];
3634
```

```
3635            color c;
3636            c.r = 255;
3637            c.g = c.b = i;
3638
3639            for (i=0; i<=2; i++) {
3640
3641                    c.r = c.r - i*20 ;
3642                    c.g = c.g + i*60;
3643                    c.b = c.b + i*60;
3644
3645                    t.colour = c;
3646
3647                    coor[i].x = i*150;
3648                    coor[i].y = 200;
3649
3650                    t.rotation = i*40;
3651
3652                    t.position = coor[i];
3653
3654                    if(i==1)
3655                            t.font = "Times.ttf";
3656
3657                    im <~ t;
3658
3659            }
3660
3661
3662    drawline(im, coor[0], coor[2], c, 5);
3663
3664    save(im, "test5.png");
3665
3666    return 0;
3667
3668    }
3669    /********
3670    * File: test3.good.txt
3671    ***********/
3672
3673    //**************************** TEST 3 *******************************
3674
3675    void zigzag2(image im, color c, int interval){
3676
3677            coordinate p1,p2;
3678
3679            int i;
3680
3681            p1.x = p2.y = 1;
3682
3683            p1.y = interval;
3684            p2.x = im.w -1 ;
3685
3686            for(i=0;i<im.h/interval;i++){
3687
3688                    if(i%2>0)
3689                            p2.y = p1.y + interval;
3690                    else
```

```
3691                            p1.y = p2.y + interval;
3692
3693                    drawline(im,p1,p2,c,3);
3694
3695            }
3696
3697            p1.x = 10;
3698            p1.y = im.w - 1;
3699
3700            p2.x = p1.x + interval;
3701            p2.y = 1;
3702
3703            for(i=0;i<im.w/interval;i++){
3704
3705                    if(i%2>0)
3706                            p1.x = p2.x + interval;
3707                    else
3708                            p2.x = p1.x + interval;
3709
3710                    drawline(im,p1,p2,c,3);
3711
3712            }
3713
3714    }
3715
3716
3717
3718    int main() {
3719
3720            color white, black;
3721
3722            white.r = white.g = white.b = 255;
3723            black.r = black.r = black.r = 0;
3724
3725            coordinate p1;
3726
3727            p1.x = 20;
3728            p1.y = 130;
3729
3730            image slashdot;
3731
3732            create(slashdot,300,700,white);
3733
3734            text w1;
3735
3736            w1.font =  "CUrLZ___.ttf";
3737            w1.colour = black;
3738            w1.name = "yqrmxas";
3739            w1.position = p1;
3740            w1.size = 160;
3741            w1.rotation = -12;
3742
3743            slashdot <~ w1;
3744
3745            zigzag2(slashdot, black, 60);
3746
```

```
3747          save(slashdot, "test3.png");
3748
3749          return 0;
3750
3751  }
3752  /********
3753  * File: test9.good.txt
3754  ***********/
3755  //*********************************  TEST 9 **********************
3756
3757  int main(){
3758
3759          image im;
3760
3761          string name;
3762
3763          text w1;
3764          w1.name = "amazing clip";
3765          w1.font = "CURLZ___.ttf";
3766          w1.rotation = 0;
3767          w1.size = 100;
3768          w1.position.x = 70;
3769          w1.position.y = 110;
3770          w1.colour.r = w1.colour.g = w1.colour.b = 255;
3771
3772
3773          for(int j = 0; j<124; j++){
3774
3775                  name = "./lake/lake" + int2string(j) + ".jpg";
3776                  open(im, name);
3777
3778                  w1.colour.b = w1.colour.b - 2;
3779                  w1.colour.g = 80;;
3780                  w1.colour.r = 0 + j*2;
3781
3782                  im <~ w1;
3783
3784                  name = "./lake2/lake" + int2string(j) + ".jpg";
3785
3786                  save(im,name);
3787
3788
3789          }
3790
3791          return 0;
3792
3793  }
3794
3795
3796  /*****************************
3797  * Directory: tests/lexer_test
3798  *****************************/
3799
3800
3801  /********
3802  * File: comments.good.txt
```

```
3803  ***********/
3804  // test comments
3805
3806  int main() {
3807
3808  int a;
3809
3810  // a=mao;
3811
3812  /*  nfasnfos
3813  fdsdfvs
3814  desf */
3815
3816  /* // sdfs //// */
3817
3818  return 0;
3819
3820
3821
3822  }
3823  /********
3824  * File: comments1.bad.txt
3825  ***********/
3826  // test comments
3827
3828  int main() {
3829
3830          int a;
3831
3832          /* ***/* a = 3 */
3833
3834
3835          return 0;
3836
3837
3838  }
3839  /********
3840  * File: comments3.bad.txt
3841  ***********/
3842  / test comments
3843
3844  int main() {
3845
3846  return 0;
3847
3848
3849
3850  }
3851  /********
3852  * File: comments2.bad.txt
3853  ***********/
3854  // test comments
3855
3856  int main() {
3857
3858          int a;
```

```
3859
3860
3861            */
3862
3863            return 0;
3864
3865
3866    }
3867    /********
3868    * File: keyword as id.bad.txt
3869    ***********/
3870    // use keyword as ID
3871
3872    int main()
3873    {
3874
3875            float e[10];
3876
3877            image coordinate;
3878
3879            color c;
3880
3881            c.r = c.b = c.g = 0;
3882
3883            create(coordinate, 50 ,90, c);
3884
3885            return 0;
3886
3887
3888    }
3889
3890    /*****************************
3891    * Directory: tests/parser_tests
3892    *****************************/
3893
3894
3895    /********
3896    * File: array_access1.bad.txt
3897    ***********/
3898    // array access problem
3899
3900    int main()
3901    {
3902
3903            float e[10];
3904            char a;
3905            e[a] = 2.5;
3906
3907            return 0;
3908
3909
3910    }
3911    /********
3912    * File: control_flow3.bad.txt
3913    ***********/
3914    // test control flow
```

```
3915
3916
3917  int foo(int x) {
3918
3919          return x - 1;
3920  }
3921
3922  int main(){
3923
3924          fo(int x=3; x<5 ;x++){
3925
3926                  aInt++;
3927
3928          }
3929
3930          for(int y=3;y!=5;y=y+2) {
3931
3932                  aInt++;
3933
3934          }
3935
3936          return 0;
3937
3938
3939  }
3940  /*********
3941  * File: control_flow.good.txt
3942  ************/
3943  // test control flow
3944
3945
3946  int foo(int x) {
3947
3948          return x - 1;
3949  }
3950
3951  int main(){
3952
3953          bool aBool ;
3954          char aChar;
3955          color aColor ;
3956          float aFloat, bFloat ;
3957          coordinate aCoor, bCoor;
3958          string aString, bString;
3959          text aText;
3960          image aImage;
3961          int aInt, bInt, cInt;
3962
3963          text arr[3];
3964
3965
3966          aInt = 2;
3967
3968
3969          // nested if
3970
```

```
3971            if(aInt) {
3972
3973                    aBool = 0;
3974            }
3975            else {
3976
3977                    if(aInt>3){
3978
3979                    }
3980            }
3981
3982
3983            // while and call to function
3984
3985            while(aInt>0){
3986
3987                    aInt = foo(aInt);
3988
3989                    if(aInt==1){
3990
3991                            break;
3992
3993                    }
3994
3995            }
3996
3997
3998            // for
3999
4000            for(int x=3;x<5;x++){
4001
4002                    aInt++;
4003
4004            }
4005
4006            for(int y=3;y!=5;y=y+2) {
4007
4008                    aInt++;
4009
4010            }
4011
4012            return 0;
4013
4014
4015    }
4016    /*********
4017    * File: fun_error2.bad.txt
4018    ************/
4019    // function error 2 : wrong return type
4020
4021
4022    float foo(int x){return x};
4023
4024    int main() {
4025
4026            int a, b, c;
```

```
4027
4028            float a = foo(3);
4029
4030            return 0;
4031
4032    }
4033
4034    /********
4035    * File: nested_expressions.good.txt
4036    ***********/
4037    // test nested expressions
4038
4039
4040    int main(){
4041
4042            bool x, y, z;
4043            int r,s,t;
4044
4045            float c;
4046
4047
4048            x = y=z = 1;
4049
4050            c = 3.4;
4051
4052            c = ( (((r+3*s) %t) * (r+4)) /3);
4053
4054            bool a = x||((y&&z)||x);
4055
4056            return 0;
4057
4058
4059
4060    }
4061    /********
4062    * File: declarations1.bad.txt
4063    ***********/
4064    // test declarations without assignments
4065
4066    int main()
4067    {
4068
4069
4070            char b;
4071            color d;
4072            float e;
4073            coordinate f;
4074            string g;
4075            text h;
4076            image i;
4077            int l,m ,n;
4078
4079            text o[3];
4080
4081            bool ;
4082
```

```
4083            return 0;
4084    }
4085    /********
4086    * File: array_access2.bad.txt
4087    ***********/
4088    // array access problem : non integer index
4089
4090    int main()
4091    {
4092
4093            float e[10];
4094
4095            coordinate p1;
4096            p1.x = p1.y = 2;
4097
4098            e[p1] = 2.5;
4099
4100            return 0;
4101
4102
4103    }
4104    /********
4105    * File: control_flow1.bad.txt
4106    ***********/
4107    // test control flow
4108
4109
4110    int foo(int x) {
4111
4112            return x - 1;
4113    }
4114
4115    int main(){
4116
4117            bool aBool ;
4118            char aChar;
4119            color aColor ;
4120            float aFloat, bFloat ;
4121            coordinate aCoor, bCoor;
4122            string aString, bString;
4123            text aText;
4124            image aImage;
4125            int aInt, bInt, cInt;
4126
4127            text arr[3];
4128
4129
4130            aInt = 2;
4131
4132
4133            // nested if
4134
4135            if(aInt) {
4136
4137                    aBool = 0;
4138            }
```

```
4139            else {
4140
4141                    if(aInt>3){
4142
4143                            continue;
4144
4145                    }
4146            }
4147
4148
4149            // while and call to function
4150
4151            while(aInt>0 {
4152
4153                    aInt = foo(aint);
4154
4155                    if(aInt=-=1){
4156
4157                            break;
4158
4159                    }
4160
4161            }
4162
4163
4164            // for
4165
4166            for(int x=3;x<5;x++){
4167
4168                    aInt++;
4169
4170            }
4171
4172            for(int y=3;y!=5;y=y+2) {
4173
4174                    aInt++;
4175
4176            }
4177
4178            return 0;
4179
4180
4181    }
4182    /*********
4183    * File: fun_error1.bad.txt
4184    ************/
4185    // function error 1 : missing return type
4186
4187
4188    foo(int x){return x};
4189
4190    int main() {
4191
4192            int a, b, c;
4193
4194            a = foo(3);
```

```
4195
4196            return 0;
4197
4198    }
4199
4200    /********
4201    * File: mismatch_parenthesis_3.bad.txt
4202    ***********/
4203    // test mismatching parentheses 3
4204
4205
4206    int main() {
4207
4208            int a=2;
4209
4210            if(a){
4211
4212                    a = 4;
4213
4214            return 0;
4215
4216    }
4217    /********
4218    * File: mismatch_parenthesis_2.bad.txt
4219    ***********/
4220    // test mismatching parentheses 2
4221
4222
4223    int main main() {
4224
4225            int a=2;
4226
4227            if(a==2 {
4228
4229                    a=3;
4230            }
4231
4232            return 0;
4233
4234    }
4235    /********
4236    * File: declarations.good.txt
4237    ***********/
4238    // test declarations without assignments
4239
4240    int main()
4241    {
4242
4243            bool a;
4244            char b;
4245            color d;
4246            float e;
4247            coordinate f;
4248            string g;
4249            text h;
4250            image i;
```

```
4251            int l, m ,n;
4252
4253            text o[3];
4254
4255            return 0;
4256    }
4257    /********
4258    * File: mismatch_parenthesis_1.bad.txt
4259    ***********/
4260    // test mismatching parentheses 1
4261
4262
4263    int main() {
4264
4265            int a=2;
4266
4267            int b = (a + 0));
4268
4269            return 0;
4270
4271    }
4272    /********
4273    * File: global_variables.good.txt
4274    ***********/
4275    // test global variables
4276
4277
4278    int a=2;
4279
4280    int main(){
4281
4282            int e = 3;
4283
4284            return a+e;
4285
4286    }
4287    /********
4288    * File: declarations_plus_assignments.good.txt
4289    ***********/
4290    // declaration  plus assignments
4291
4292    int main()
4293    {
4294
4295            bool a = false;
4296            char b = 'e';
4297            color d ;
4298            float e = 5.8;
4299            coordinate f;
4300            string g = "Hello world";
4301            text h;
4302            image i;
4303            int l,m ,n;
4304
4305            text o[3];
4306
```

```
4307            d.r = 255;
4308
4309            h.name = "Hi";
4310            h.name = g;
4311
4312            h.font = "Times.ttf";
4313
4314            h.colour.r = 255;
4315            h.colour.g = d.b;
4316
4317            f.x = 2;
4318
4319            h.position.x = 40;
4320
4321            h.rotation = 34;
4322
4323            h.size = 7;
4324
4325            o[0].name = "Hello";
4326            o[1].colour = d;
4327
4328
4329            int p = l;
4330
4331            color d1;
4332
4333            d1.r = d1.g = d1.b = 25;
4334
4335            return 0;
4336
4337    }
4338    /*********
4339    * File: fun_declaration_plus_overloading.good.txt
4340    ************/
4341    // test function declarations and overloading
4342
4343    int foo(bool a, coordinate b)
4344    {
4345            return b.x;
4346    }
4347
4348    int foo(bool a)
4349    {
4350            return 1;
4351    }
4352
4353
4354    int main(){
4355
4356
4357            int a;
4358            bool b = true;
4359            coordinate c;
4360
4361            c.x = foo(b);
4362
```

```
4363            a = foo(b,c);
4364
4365            return 0;
4366
4367    }
4368
4369    /********
4370    * File: missing_element_2.bad.txt
4371    ***********/
4372    // missing element comma
4373
4374    int  main() {
4375
4376            inty a, b c;
4377
4378            return 0;
4379
4380
4381    }
4382
4383    /********
4384    * File: declarations3.bad.txt
4385    ***********/
4386    // test declarations without assignments
4387
4388    int main()
4389    {
4390
4391
4392            cthar b;
4393
4394
4395            return 0;
4396    }
4397    /********
4398    * File: array_access3.bad.txt
4399    ***********/
4400    // array access problem : non integer index
4401
4402    int main()
4403    {
4404
4405            float e[10];
4406
4407            image im;
4408
4409            e[im] = 2.5;
4410
4411            return 0;
4412
4413
4414    }
4415    /********
4416    * File: declarations2.bad.txt
4417    ***********/
4418    // test declarations without assignments
```

```
4419
4420  int main()
4421  {
4422
4423
4424          char b;
4425          color d;
4426          float e;
4427          coordinate f;
4428          string g;
4429          text h;
4430          image i;
4431          int l,m ,n;
4432
4433          text o[3;
4434
4435
4436          return 0;
4437  }
4438  /********
4439  * File: array_access.good.txt
4440  ***********/
4441  // use keyword as ID
4442
4443  int main()
4444  {
4445
4446          int e[10];
4447
4448          image a[10];
4449
4450          color c;
4451
4452          c.r = c.g = c.g = 20;
4453
4454          for(int i= 0; i<10; i++){
4455                  e[i] = i;
4456
4457                  create(a[i],e[i],e[i] +20, c);
4458          }
4459
4460          return 0;
4461
4462
4463  }
4464  /********
4465  * File: fun_error3.bad.txt
4466  ***********/
4467  // function error 3 : wrong return assignment type
4468
4469
4470  float foo(int x){return 4.04};
4471
4472  int main() {
4473
4474          bool x = foo(3);
```

```
4475
4476            return 0;
4477
4478    }
4479    /********
4480    * File: operators_2.good.txt
4481    ***********/
4482    // test operators 2
4483
4484
4485
4486    int main()
4487    {
4488
4489            bool aBool, bBool ;
4490            char aChar, bChar;
4491            color aColor, bColor ;
4492            float aFloat, bFloat ;
4493            coordinate aCoor, bCoor;
4494            string aString, bString = "try.txt";
4495            text aText ;
4496            image aImage;
4497            int aInt, bInt, cInt;
4498
4499            text arr[3];
4500
4501            aInt = 4;
4502
4503            aCoor.y = 7;
4504
4505
4506            // euquality comparisons
4507
4508
4509            if(aInt==5) {
4510
4511                    aBool = true;
4512
4513            }
4514
4515
4516            if(aBool) {
4517
4518                    if(aCoor.y != 3) {
4519
4520                            bInt = 3;
4521
4522                    }
4523            }
4524
4525
4526            // relational comparisons
4527
4528            bBool = (bInt<=3);
4529
4530
```

```
4531          // logical AND and OR and relational comparisons
4532
4533          aBool = (aInt>5)&&(aInt<10)||3>=2&&aCoor.y<=7;
4534
4535
4536          // open
4537
4538          open(aImage, bString);
4539
4540          aText.name = "Hi";
4541          aText.font = "Arial.ttf";
4542          aText.colour.r = 255;
4543          aText.position.x = 40;
4544          aText.rotation = 34;
4545          aText.size = 7;
4546
4547          // <~
4548
4549          aImage <~ aText;
4550
4551
4552          return 0;
4553
4554
4555  }
4556  /********
4557  * File: control_flow2.bad.txt
4558  ***********/
4559  // test control flow
4560
4561
4562  int foo(int x) {
4563
4564          return x - 1;
4565  }
4566
4567  int main(){
4568
4569          for(int x=3 x<5;x++){
4570
4571                  aInt++;
4572
4573          }
4574
4575          for(int y=3;y!=5;y=y+2) {
4576
4577                  aInt++;
4578
4579          }
4580
4581          return 0;
4582
4583
4584  }
4585  /********
4586  * File: missing_element_1.bad.txt
```

```
4587  ***********/
4588  // missing element semicolon
4589
4590  int main() {
4591
4592          inty a, b, c;
4593
4594          c = 2
4595
4596          return 0;
4597
4598  }
4599
4600  /********
4601  * File: operators_1.good.txt
4602  ***********/
4603  // test operators 1
4604
4605
4606
4607  int main()
4608  {
4609
4610          bool aBool, bBool ;
4611          char aChar, bChar;
4612          color aColor, bColor ;
4613          float aFloat, bFloat ;
4614          coordinate aCoor, bCoor;
4615          string aString, bString;
4616          text aText;
4617          image aImage;
4618          int aInt, bInt, cInt;
4619
4620          text arr[3];
4621
4622          // assignment
4623
4624          aBool = true;
4625          bBool = aBool;
4626          aBool = bBool = false;
4627
4628          aChar = 'y';
4629          bChar = aChar;
4630
4631          aColor.r = 255;
4632          bColor.g = aColor.b = 3;
4633
4634
4635          aFloat = 7;
4636          bFloat = -3.544;
4637          aFloat = bFloat;
4638
4639          aCoor.x = 37;
4640          bCoor = aCoor;
4641
4642          aString = "Hi";
```

```
4643            aString = "Hello";
4644            bString = aString;
4645
4646
4647            aText.name = "Hi";
4648            aText.name = bString;
4649            aText.font = "Times.ttf";
4650            aText.colour.r = 255;
4651            aText.colour.g = bColor.b;
4652            aText.position.x = 40;
4653            aText.position.x = aText.position.y = 245;
4654            aText.position = aCoor;
4655            aText.rotation = 34;
4656            aText.rotation = aInt;
4657            aText.size = 7;
4658            aText.size = bInt;
4659
4660            aInt = bInt = cInt = 3;
4661
4662            arr[0].name = "Hello";
4663            arr[1].colour = bColor;
4664
4665
4666            // increment and decrement
4667
4668            aInt++;
4669            aText.position.x-- ;
4670
4671
4672            // plus and minus
4673
4674            aInt = bInt + 7;
4675            bFloat = bFloat - aFloat + 4;
4676
4677
4678            // logical not
4679
4680            aBool = !bBool;
4681
4682
4683            // sign operator
4684
4685            aInt = -bInt;
4686
4687
4688            // multiply - divide - modulus
4689
4690            aFloat = aText.position.y * aText.rotation / 2 % 4;
4691
4692            return 0;
4693
4694
4695    }
4696
4697    /*****************************
4698    * Directory: tests/walking_tests
```

```
4699  ******************************/
4700
4701
4702  /********
4703  * File: expr.equal.bad.2.txt
4704  ***********/
4705  // expr equal
4706
4707  int main()
4708  {
4709          int i1, i2;
4710          float f1, f2;
4711          char c1, c2;
4712          string s1, s2;
4713          coordinate co1, co2;
4714          color col1, col2;
4715          bool b1, b2;
4716          i1==i2;
4717          f1==f2;
4718          c1!=c2;
4719          i1!=c1;
4720          i1==f1;
4721          s1==s2;
4722          b1==b2;
4723  }
4724  /********
4725  * File: expr.arith.bad.4.txt
4726  ***********/
4727  // expr arith
4728
4729  int main()
4730  {
4731          int i1, i2;
4732          float f1, f2;
4733          char c1, c2;
4734          string s1, s2;
4735          coordinate co1, co2;
4736          color col1, col2;
4737          i1+i2;
4738          i1+f1;
4739          f1+f2;
4740          i1+c1;
4741          c1+c2;
4742          co1+co2;
4743          col1+col2;
4744          s1+c2;
4745          s1+s2;
4746
4747          f1+s1;
4748  }
4749  /********
4750  * File: break.bad.2.txt
4751  ***********/
4752  // Break, continue
4753
4754  int main()
```

```
4755  {
4756          int a;
4757          {
4758                  while (1)
4759                  {
4760                  }
4761                  break;
4762          }
4763
4764          return a;
4765  }
4766  /********
4767  * File: expr.unary.bad.4.txt
4768  ***********/
4769  // expr unary
4770
4771  int main()
4772  {
4773          int i1, i2;
4774          float f1, f2;
4775          char c1, c2;
4776          string s1, s2;
4777          coordinate co1, co2;
4778          color col1, col2;
4779          bool b1, b2;
4780          +col1;
4781  }
4782  /********
4783  * File: expr.arith.bad.5.txt
4784  ***********/
4785  // expr arith
4786
4787  int main()
4788  {
4789          int i1, i2;
4790          float f1, f2;
4791          char c1, c2;
4792          string s1, s2;
4793          coordinate co1, co2;
4794          color col1, col2;
4795          i1+i2;
4796          i1+f1;
4797          f1+f2;
4798          i1+c1;
4799          c1+c2;
4800          co1+co2;
4801          col1+col2;
4802          s1+c2;
4803          s1+s2;
4804
4805          col1+co1;
4806  }
4807  /********
4808  * File: expr.logic.bad.2.txt
4809  ***********/
4810  // expr logic
```

```
4811
4812  int main()
4813  {
4814          int i1, i2;
4815          float f1, f2;
4816          char c1, c2;
4817          string s1, s2;
4818          coordinate co1, co2;
4819          color col1, col2;
4820          bool b1, b2;
4821          c2||b1;
4822  }
4823  /*********
4824  * File: expr.arith.bad.7.txt
4825  ************/
4826  // expr arith
4827
4828  int main()
4829  {
4830          int i1, i2;
4831          float f1, f2;
4832          char c1, c2;
4833          string s1, s2;
4834          coordinate co1, co2;
4835          color col1, col2;
4836          bool b1, b2;
4837          i1+i2;
4838          i1+f1;
4839          f1+f2;
4840          i1+c1;
4841          c1+c2;
4842          co1+co2;
4843          col1+col2;
4844          s1+c2;
4845          s1+s2;
4846
4847          s1-s2;
4848  }
4849  /*********
4850  * File: expr.arith.bad.2.txt
4851  ************/
4852  // expr arith
4853
4854  int main()
4855  {
4856          int i1, i2;
4857          float f1, f2;
4858          char c1, c2;
4859          string s1, s2;
4860          coordinate co1, co2;
4861          color col1, col2;
4862          i1+i2;
4863          i1+f1;
4864          f1+f2;
4865          i1+c1;
4866          c1+c2;
```

```
4867            co1+co2;
4868            col1+col2;
4869            s1+c2;
4870            s1+s2;
4871
4872            i1+col1;
4873    }
4874    /********
4875    * File: scope.bad.1.txt
4876    ***********/
4877    // Entering and leaving scope
4878
4879
4880    int main()
4881
4882    {
4883
4884            int a;
4885
4886            {
4887                    b = b + 2;
4888            }
4889
4890            return a;
4891    }
4892    /********
4893    * File: scope.good.1.txt
4894    ***********/
4895    // Entering and leaving scope
4896
4897
4898    int main()
4899
4900    {
4901
4902            int a;
4903
4904            {
4905                    a = a + 2;
4906            }
4907
4908            return a;
4909    }
4910    /********
4911    * File: continue.bad1.txt
4912    ***********/
4913    // Break, continue
4914
4915    int main()
4916    {
4917            int a;
4918            {
4919                    continue;
4920            }
4921
4922            return a;
```

```
4923  }
4924  /********
4925  * File: continue.good.1.txt
4926  ***********/
4927  // Break, continue
4928
4929  int main()
4930  {
4931          int a;
4932          {
4933                  while (1)
4934                  {
4935                          continue;
4936                  }
4937          }
4938
4939          return a;
4940  }
4941  /********
4942  * File: break.good.1.txt
4943  ***********/
4944  // Break, continue
4945
4946  int main()
4947  {
4948          int a;
4949          {
4950                  while (1)
4951                  {
4952                          break;
4953                  }
4954          }
4955
4956          return a;
4957  }
4958  /********
4959  * File: expr.equal.bad.1.txt
4960  ***********/
4961  // expr equal
4962
4963  int main()
4964  {
4965          int i1, i2;
4966          float f1, f2;
4967          char c1, c2;
4968          string s1, s2;
4969          coordinate co1, co2;
4970          color col1, col2;
4971          bool b1, b2;
4972          i1==i2;
4973          f1==f2;
4974          c1!=c2;
4975          i1!=c1;
4976          i1==f1;
4977          s1==s2;
4978          col1==col2;
```

```
4979  }
4980  /********
4981  * File: continue.bad.2.txt
4982  ***********/
4983  // Break, continue
4984
4985  int main()
4986  {
4987          int a;
4988          {
4989                  while (1)
4990                  {
4991                  }
4992                  continue;
4993          }
4994
4995          return a;
4996  }
4997  /********
4998  * File: expr.arith.bad.1.txt
4999  ***********/
5000  // expr arith
5001
5002  int main()
5003  {
5004          int i1, i2;
5005          float f1, f2;
5006          char c1, c2;
5007          string s1, s2;
5008          coordinate co1, co2;
5009          color col1, col2;
5010          i1+i2;
5011          i1+f1;
5012          f1+f2;
5013          i1+c1;
5014          c1+c2;
5015          co1+co2;
5016          col1+col2;
5017          s1+c2;
5018          s1+s2;
5019
5020          i1+s1;
5021  }
5022  /********
5023  * File: expr.logic.bad.1.txt
5024  ***********/
5025  // expr logic
5026
5027  int main()
5028  {
5029          int i1, i2;
5030          float f1, f2;
5031          char c1, c2;
5032          string s1, s2;
5033          coordinate co1, co2;
5034          color col1, col2;
```

```
5035            bool b1, b2;
5036            i1++;
5037            f1++;
5038            c1++;
5039    }
5040    /********
5041    * File: expr.unary.good.2.txt
5042    ***********/
5043    // expr unary
5044
5045    int main()
5046    {
5047            int i1, i2;
5048            float f1, f2;
5049            char c1, c2;
5050            string s1, s2;
5051            coordinate co1, co2;
5052            color col1, col2;
5053            bool b1, b2;
5054            +1;
5055            -1;
5056            -i1;
5057            -c1;
5058    }
5059    /********
5060    * File: expr.unary.good.1.txt
5061    ***********/
5062    // expr unary
5063
5064    int main()
5065    {
5066            int i1, i2;
5067            float f1, f2;
5068            char c1, c2;
5069            string s1, s2;
5070            coordinate co1, co2;
5071            color col1, col2;
5072            bool b1, b2;
5073            i1++;
5074            f1++;
5075            c1++;
5076    }
5077    /********
5078    * File: func.bad.1.txt
5079    ***********/
5080    // func
5081
5082    int main()
5083    {
5084            foo();
5085            return 1;
5086    }
5087
5088    int foo()
5089    {
5090            return 1;
```

```
5091  }
5092  /********
5093  * File: buildinfunc.good.1.txt
5094  ***********/
5095  // func
5096
5097  int main()
5098  {
5099          color col;
5100          image im;
5101          char c;
5102          string s;
5103          int i;
5104          float f;
5105          col.r = col.g = col.b = 30;
5106          coordinate co1, co2;
5107          create(im, 2, 2, col);
5108          drawline(im, co1, co2, col, 2);
5109          save(im, "1");
5110          open(im, "1");
5111          c = char_at(s,1);
5112          s = int2string(i);
5113          s = float2string(f);
5114          i = string2int(s);
5115          f = string2float(s);
5116          return 1;
5117  }
5118
5119  /********
5120  * File: expr.equal.good.1.txt
5121  ***********/
5122  // expr equal
5123
5124  int main()
5125  {
5126          int i1, i2;
5127          float f1, f2;
5128          char c1, c2;
5129          string s1, s2;
5130          coordinate co1, co2;
5131          color col1, col2;
5132          bool b1, b2;
5133          i1==i2;
5134          f1==f2;
5135          c1!=c2;
5136          i1!=c1;
5137          i1==f1;
5138          s1==s2;
5139  }
5140  /********
5141  * File: expr.compr.bad.1.txt
5142  ***********/
5143  // expr arith
5144
5145  int main()
5146  {
```

```
5147            int i1, i2;
5148            float f1, f2;
5149            char c1, c2;
5150            string s1, s2;
5151            coordinate co1, co2;
5152            color col1, col2;
5153            bool b1, b2;
5154            i1<i2;
5155            f1>f2;
5156            c1<=c2;
5157
5158            s1<s2;
5159    }
5160    /********
5161    * File: scope.bad.2.txt
5162    ***********/
5163    // Entering and leaving scope
5164
5165
5166    int main()
5167
5168    {
5169
5170            int a;
5171
5172            {
5173                    int b;
5174                    b = b + 2;
5175            }
5176            b = b + 2;
5177            return a;
5178    }
5179    /********
5180    * File: break.bad.1.txt
5181    ***********/
5182    // Break, continue
5183
5184    int main()
5185    {
5186            int a;
5187            {
5188                    break;
5189            }
5190
5191            return a;
5192    }
5193    /********
5194    * File: expr.logic.good.1.txt
5195    ***********/
5196    // expr logic
5197
5198    int main()
5199    {
5200            int i1, i2;
5201            float f1, f2;
5202            char c1, c2;
```

```
5203            string s1, s2;
5204            coordinate co1, co2;
5205            color col1, col2;
5206            bool b1, b2;
5207            b1||b2;
5208            b1&&b2;
5209            !b2;
5210    }
5211    /********
5212    * File: scope.good.2.txt
5213    ***********/
5214    // Entering and leaving scope
5215
5216
5217    int main()
5218    {
5219            int a;
5220            {
5221                    int b;
5222                    {
5223                            b = b + 2;
5224                    }
5225            }
5226
5227            a = a + 2;
5228            return a;
5229    }
5230    /********
5231    * File: expr.compr.bad.3.txt
5232    ***********/
5233    // expr compr
5234
5235    int main()
5236    {
5237            int i1, i2;
5238            float f1, f2;
5239            char c1, c2;
5240            string s1, s2;
5241            coordinate co1, co2;
5242            color col1, col2;
5243            bool b1, b2;
5244            i1<i2;
5245            f1>f2;
5246            c1<=c2;
5247
5248            b1<b2;
5249    }
5250    /********
5251    * File: buildinfunc.bad.1.txt
5252    ***********/
5253    // func
5254
5255    int main()
5256    {
5257            color col;
5258            image im;
```

```
5259            char c;
5260            string s;
5261            int i;
5262            float f;
5263            col.r = col.g = col.b = 30;
5264            coordinate co1, co2;
5265            create(im, 2, 2, col);
5266            drawline(im, co1, co2, col, 2);
5267            save(im, "1");
5268            save(im, c);
5269            open(im, "1");
5270            c = char_at(s,1);
5271            s = int2string(i);
5272            s = float2string(f);
5273            i = string2int(s);
5274            f = string2float(s);
5275            return 1;
5276    }
5277
5278    /********
5279    * File: expr.unary.bad.1.txt
5280    ***********/
5281    // expr unary
5282
5283    int main()
5284    {
5285            int i1, i2;
5286            float f1, f2;
5287            char c1, c2;
5288            string s1, s2;
5289            coordinate co1, co2;
5290            color col1, col2;
5291            bool b1, b2;
5292            b2++;
5293    }
5294    /********
5295    * File: func.good.1.txt
5296    ***********/
5297    // func
5298
5299    int foo()
5300    {
5301            return 1;
5302    }
5303    int main()
5304    {
5305            foo();
5306            return 1;
5307    }
5308    /********
5309    * File: scope.bad.3.txt
5310    ***********/
5311    // Entering and leaving scope
5312
5313
5314    int main()
```

```
5315  {
5316          int a;
5317          {
5318                  int b;
5319                  {
5320                          b = b + 2;
5321                  }
5322          }
5323
5324          b = b + 2;
5325          return a;
5326  }
5327  /*********
5328  * File: func.bad.2.txt
5329  ************/
5330  // func
5331
5332  int main()
5333  {
5334          foo();
5335          return 1;
5336  }
5337
5338  /*********
5339  * File: scope.good.3.txt
5340  ************/
5341  // Entering and leaving scope
5342  int b;
5343
5344  int main()
5345  {
5346          int a;
5347          {
5348                  b = b + 2;
5349          }
5350
5351          a = a + 2;
5352          return a;
5353  }
5354  /*********
5355  * File: expr.equal.bad.3.txt
5356  ************/
5357  // expr equal
5358
5359  int main()
5360  {
5361          int i1, i2;
5362          float f1, f2;
5363          char c1, c2;
5364          string s1, s2;
5365          coordinate co1, co2;
5366          color col1, col2;
5367          bool b1, b2;
5368          i1==i2;
5369          f1==f2;
5370          c1!=c2;
```

```
5371            i1!=c1;
5372            i1==f1;
5373            s1==s2;
5374            c1==s2;
5375    }
5376    /********
5377    * File: scope.bad.4.txt
5378    ***********/
5379    // Entering and leaving scope
5380
5381
5382    int main()
5383    {
5384            int a;
5385            {
5386                    int b;
5387                    {
5388                            b = b + 2;
5389                    }
5390
5391            }
5392            {
5393                    b = b + 2;
5394            }
5395            return a;
5396    }
5397    /********
5398    * File: expr.arith.bad.6.txt
5399    ***********/
5400    // expr arith
5401
5402    int main()
5403    {
5404            int i1, i2;
5405            float f1, f2;
5406            char c1, c2;
5407            string s1, s2;
5408            coordinate co1, co2;
5409            color col1, col2;
5410            bool b1, b2;
5411            i1+i2;
5412            i1+f1;
5413            f1+f2;
5414            i1+c1;
5415            c1+c2;
5416            co1+co2;
5417            col1+col2;
5418            s1+c2;
5419            s1+s2;
5420
5421            b1+b2;
5422    }
5423    /********
5424    * File: expr.logic.bad.3.txt
5425    ***********/
5426    // expr logic
```

```
5427
5428  int main()
5429  {
5430          int i1, i2;
5431          float f1, f2;
5432          char c1, c2;
5433          string s1, s2;
5434          coordinate co1, co2;
5435          color col1, col2;
5436          bool b1, b2;
5437          i1||b1;
5438  }
5439  /*********
5440  * File: expr.arith.bad.3.txt
5441  ************/
5442  // expr arith
5443
5444  int main()
5445  {
5446          int i1, i2;
5447          float f1, f2;
5448          char c1, c2;
5449          string s1, s2;
5450          coordinate co1, co2;
5451          color col1, col2;
5452          i1+i2;
5453          i1+f1;
5454          f1+f2;
5455          i1+c1;
5456          c1+c2;
5457          co1+co2;
5458          col1+col2;
5459          s1+c2;
5460          s1+s2;
5461
5462          f1+col1;
5463  }
5464  /*********
5465  * File: expr.arith.good.1.txt
5466  ************/
5467  // expr arith
5468
5469  int main()
5470  {
5471          int i1, i2;
5472          float f1, f2;
5473          char c1, c2;
5474          string s1, s2;
5475          coordinate co1, co2;
5476          color col1, col2;
5477          i1+i2;
5478          i1+f1;
5479          f1+f2;
5480          i1+c1;
5481          c1+c2;
5482          co1+co2;
```

```
5483            col1+col2;
5484            s1+c2;
5485            s1+s2;
5486    }
5487    /********
5488    * File: expr.unary.bad.3.txt
5489    ***********/
5490    // expr unary
5491
5492    int main()
5493    {
5494            int i1, i2;
5495            float f1, f2;
5496            char c1, c2;
5497            string s1, s2;
5498            coordinate co1, co2;
5499            color col1, col2;
5500            bool b1, b2;
5501            +b1;
5502    }
5503    /********
5504    * File: expr.compr.good.1.txt
5505    ***********/
5506    // expr compr
5507
5508    int main()
5509    {
5510            int i1, i2;
5511            float f1, f2;
5512            char c1, c2;
5513            string s1, s2;
5514            coordinate co1, co2;
5515            color col1, col2;
5516            bool b1, b2;
5517            i1<i2;
5518            f1>f2;
5519            c1<=c2;
5520            i1>=c1;
5521            i1>f1;
5522    }
5523    /********
5524    * File: scope.good.4.txt
5525    ***********/
5526    // Entering and leaving scope
5527
5528
5529    int main()
5530    {
5531            int a;
5532            {
5533                    int b;
5534                    {
5535                            b = b + 2;
5536                    }
5537
5538            }
```

```
5539            {
5540                    a = a + 2;
5541            }
5542            return a;
5543    }
5544    /********
5545    * File: expr.unary.bad.2.txt
5546    ***********/
5547    // expr unary
5548
5549    int main()
5550    {
5551            int i1, i2;
5552            float f1, f2;
5553            char c1, c2;
5554            string s1, s2;
5555            coordinate co1, co2;
5556            color col1, col2;
5557            bool b1, b2;
5558            1++;
5559    }
5560    /********
5561    * File: expr.compr.bad.2.txt
5562    ***********/
5563    // expr compr
5564
5565    int main()
5566
5567            int i1, i2;
5568            float f1, f2;
5569            char c1, c2;
5570            string s1, s2;
5571            coordinate co1, co2;
5572            color col1, col2;
5573            bool b1, b2;
5574            i1<i2;
5575            f1>f2;
5576            c1<=c2;
5577
5578            col1<col2;
5579    }
```

```
#ifndef TMIL_H
#define TMIL_H



#include <gd.h>
#include <gdfontl.h>
#include <gdfontt.h>
#include <gdfonts.h>
#include <gdfontmb.h>
#include <gdfontg.h>
#include <string>
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <vector>
#include <math.h>
#include <sstream>



using namespace std;

#define create(im,x,y,z) im.create_image(x,y,z)
#define open(im,s) im.open_image(s)


int max(int a, int b){return (a>b) ? a : b;}
int min(int a, int b){return (a<b) ? a : b;}


// classes for the built in types


class color
{

public:
    int r;
    int g;
    int b;



    color(){ r=0; g=0; b=0; };
    color(int R, int G, int B) { r=R; g=G; b=B;};
    int get_r(){ return r;};
    int get_g(){ return g;};
    int get_b(){ return b;};
    void set_r(int a) { r = a; };
    void set_g(int a) { g = a; };
    void set_b(int a) { b = a; };

    ~color(){};

    color& operator=(color c){ r=c.get_r(); g=c.get_g(); b=c.get_b(); return *this; };
    color operator+(color c){ int R = min(get_r() + c.get_r(),255)  ; int G = min(get_g() + c.get
_g(),255) ; int B = min(get_b() + c.get_b(),255) ; return color(R,G,B); };
    color operator-(color c){ int R = max(get_r() - c.get_r(),0) ; int G = max(get_g() - c.get_g(
),0) ; int B = max(get_b() - c.get_b(),0) ; return color(R,G,B); };
    color operator*(color c){ int R = min(get_r() * c.get_r(),255) ; int G = min(get_g() * c.get_
g(),255) ; int B = min(get_b() * c.get_b(),255) ; return color(R,G,B); };
    color operator/(color c){ int R = get_r() / c.get_r() ; int G = get_g() / c.get_g() ; int B =
 get_b() / c.get_b() ; return color(R,G,B); };
    color operator%(color c){ int R = get_r() % c.get_r() ; int G = get_g() % c.get_g() ; int B =
 get_b() % c.get_b() ; return color(R,G,B); };
    color& operator++(){ r = min(r+1,255); g = min(g+1,255); b = min(b+1,255); return *this; };
    color& operator--(){ r = max(r-1,0); g = max(g-1,0); b = max(b-1,0); return *this; };
    color& operator+=(color c){ r= min(get_r() + c.get_r(),255); g= min(get_g() + c.get_g(),255);
 b= min(get_b() + c.get_b(),255); return *this; };
    color& operator-=(color c){ r= max(get_r() - c.get_r(),0); g= max(get_g() - c.get_g(),0); b=
max(get_b() - c.get_b(),0); return *this; };
    bool operator==(color c){ return (get_r()==c.get_r() && get_g()==c.get_g() && get_b()==c.get_
```

```
b()); };
    bool operator!=(color c){ return (get_r()!=c.get_r() || get_g()!=c.get_g() || get_b()!=c.get_
b()); };


};


class coordinate
{

public:

    int x;
    int y;


    coordinate(){ x=0; y=0; };
    coordinate(int X,int Y){ x=X; y=Y;};
    int get_x(){ return x;};
    int get_y(){ return y;};
    void set_x(int a) { x = a; };
    void set_y(int a) { y = a; };

    ~coordinate(){};

    coordinate& operator=(coordinate c){ x=c.get_x(); y=c.get_y(); return *this; };
    coordinate operator+(coordinate c){ int x1 = get_x() + c.get_x() ; int y1 = get_y() + c.get_y
() ; return coordinate(x1,y1); };
    coordinate operator-(coordinate c){ int x1 = get_x() - c.get_x() ; int y1= get_y() - c.get_y(
) ; return coordinate(x1,y1); };
    coordinate operator*(coordinate c){ int x1 = get_x() * c.get_x() ; int y1 = get_y() * c.get_y
() ; return coordinate(x1,y1); };
    coordinate operator/(coordinate c){ int x1 = get_x() / c.get_x() ; int y1 = get_y() / c.get_y
() ; return coordinate(x1,y1); };
    coordinate operator%(coordinate c){ int x1 = get_x() % c.get_x() ; int y1 = get_y() % c.get_y
() ; return coordinate(x1,y1); };
    coordinate& operator++(){ x++; y++;  return *this; };
    coordinate& operator--(){ x--; y--;  return *this; };
    coordinate& operator+=(coordinate c){ x= get_x() + c.get_x(); y= get_y() + c.get_y(); return
*this; };
    coordinate& operator-=(coordinate c){ x= get_x() - c.get_x(); y= get_y() - c.get_y(); return
*this; };
    bool operator==(coordinate c){ return (get_x()==c.get_x() && get_y()==c.get_y()); };
    bool operator!=(coordinate c){ return (get_x()!=c.get_x() || get_y()!=c.get_y()); };

};



class word
{

public:
    string name;
    string font;
    color colour;
    coordinate position;
    int rotation;
    int size;


    word() { rotation = 0; size=1; };
    ~word() {};

    string get_name() { return name; };
    string get_font() { return font; };
    color get_color() { return colour; };
    coordinate get_position() { return position; };
    int get_rotation() { return rotation; };
    int get_size() { return size; };

    void set_name(string n) { name =n; };
```

```
    void set_font(string f) { font = f; };
    void set_color(color c) { colour = c; };
    void set_position(coordinate c) { position = c; };
    void set_rotation(int i) { rotation = i; };
    void set_size(int i) { size = i; };

    word& operator=(word w){ name = w.get_name() ; font = w.get_font(); colour = w.get_color(); p
osition = w.get_position(); rotation = w.get_rotation(); size = w.get_size(); return *this; };


};


class image
{

private:
    gdImagePtr imgPtr;

public:

    int h;
    int w;


    image() {h=0; w=0; imgPtr=NULL; };
    ~image(){ h=0; w=0; imgPtr=NULL;};  //gdImageDestroy(imgPtr);

    gdImagePtr get_imgPtr(){ return imgPtr; };
    int get_w(){ return w;};
    int get_h(){ return h;};

    void set_imgPtr(gdImagePtr p) { imgPtr = p; };
    void set_w(int a) { w = a; };
    void set_h(int a) { h = a; };

    bool create_image(int, int, color);
    bool open_image(string);

    image& operator=(image);

};


// additional functions fro class image


image& image::operator=(image im){

    if(imgPtr!=NULL) {
        gdImageDestroy(imgPtr);
    }

    imgPtr = gdImageCreateTrueColor(im.get_w(),im.get_h());

    gdImageCopy(imgPtr, im.get_imgPtr(),0,0,0,0, im.get_w(),im.get_h() );
    h = im.get_h();
    w = im.get_w();

    return *this;

};


bool image::create_image(int y, int x, color c)
{

    h = y;
    w = x;
    imgPtr = gdImageCreate(x, y);
    gdImageColorAllocate(imgPtr, c.get_r(), c.get_g(), c.get_b() );
    return 1;
```

```
}

bool image::open_image(string s){

    FILE * fin;

    fin = fopen(s.c_str(),"rb");

    if(s.at(s.length() - 2) == 'n'){

        imgPtr = gdImageCreateFromPng(fin);

        h = imgPtr->sy;
        w = imgPtr->sx;

    }
    else {

        imgPtr = gdImageCreateFromJpeg(fin);

        h = imgPtr->sy;
        w = imgPtr->sx;


    }

    fclose(fin);

    return 1;


}



// utility functions to save an image, draw lines and text on an image

void save(image im, string s)
{

    FILE* fout;

    fout = fopen(s.c_str(),"wb");

    if(s.at(s.length() - 2) == 'n'){

        gdImagePng(im.get_imgPtr(), fout);

    }
    else {

        gdImageJpeg(im.get_imgPtr(), fout,100);

    }

    fclose(fout);

}


void drawline(image im, coordinate p1, coordinate p2, color col, int width){

    int c = gdImageColorAllocate(im.get_imgPtr(), col.get_r(), col.get_g(), col.get_b());

    gdImageSetThickness(im.get_imgPtr(), width);

    gdImageLine(im.get_imgPtr(), p1.get_x(), p1.get_y(), p2.get_x(), p2.get_y(), c);


}
```

```
void stamp(image im, word t)
{

    gdFTUseFontConfig(1);

    int brect[8];

    int len = t.get_font().length();

    char* font = new char[len];
    font[len] = '\0';

    len = t.get_name().length();
    char* name = new char[len];
    name[len] = '\0';


    t.get_font().copy(font, t.get_font().length());
    t.get_name().copy(name, t.get_name().length());

    int fg = gdImageColorResolve(im.get_imgPtr(), t.get_color().get_r(), t.get_color().get_g(), t
.get_color().get_b() );

    gdImageStringFT( im.get_imgPtr(), &brect[0], fg, font , t.get_size(), ( (double)t.get_rotatio
n() * (2*3.141592653 /360) ) , t.get_position().get_x() , t.get_position().get_y() , name);


}


// utility for int, float and string conversions


float string2float(string s){

    float f = atof(s.c_str());

    return f;

}



int string2int(string s){

    int i = atoi(s.c_str());

    return i;

}


string int2string(int i){

    string s;
    stringstream out;
    out << i;
    s = out.str();

    return s;

}


string float2string(float f, int minN, int maxN){

    string s ;

    int pos, len, k;

    stringstream out;
```

```
    out << f;
    s = out.str();

    pos = s.find('.');

    len = s.length() - pos - 1;

    if(minN>maxN){
        maxN = minN;
    }

    if(len<minN){
        for(k= len; k<minN ; k++)
            s.append("0");
    }
    if(len>maxN){
        s.erase(pos + maxN + 1,(maxN-len));
    }

    return s;

}


char char_at(string s, int pos){

    return s[pos];

}




#endif //TMIL_H
```